AltiVec Programming Model

REVISION 1.2.3

Revision 0.1 February 3, 1997
Revision 0.2 February 10, 1997
Revision 0.3 March 7, 1997
Revision 0.3.1 March 24, 1997
Revision 0.3.2 March 27, 1997
Revision 1.0 April 14, 1997
Revision 1.1 June 3, 1997
Revision 1.1.1 July 2, 1997
Revision 1.1.2 September 8, 1997
Revision 1.1.3 October 2, 1997
Revision 1.1.4 March 4, 1998
Revision 1.1.5 March 4, 1998
Revision 1.1.6 March 4, 1998
Revision 1.1.7 April 6, 1998
Revision 1.1.8 April 16, 1998
Revision 1.2.0 April 30, 1998
Revision 1.2.1 May 8, 1998
Revision 1.2.2 June 4, 1998
Revision 1.2.3 September 22, 1998

Note: Version 1.2 of the AltiVec Programming Model is intended to be consistent with the AltiVec specification 1.2.3 (7/18/97). The AltiVec Programming Model replaces section 6 "Software Considerations" of the AltiVec specification.

Please send all comments to Mike Phillip (R12869@email.sps.mot.com) or Roger Smith (RZXE50@email.sps.mot.com) or Tom Wood (RA0417@email.sps.mot.com)

1. Introduction

This document defines a programming model for use with the AltiVec instruction set extension to the PowerPC architecture. There are three types of programming interfaces described in this document:

- A high-level language interface, intended for use within programming languages such as C or C++
- An application binary interface (ABI) defining low-level coding conventions
- An assembly language interface

Although a higher-level application programming interface (API) such as "mediaLib" is intended for use with AltiVec, such a specification is not addressed by this document.

1.1 Versions

This document, the corresponding C++ class library, and the Motorola AltiVec-enabled C compiler all incorporate information about AltiVec from a common source. The version number associated with the programming model will be of the form `v.r`. Corresponding versions of the C++ class library will have version number `v.r.n`, where the `n` allows corrections to be made without underlying changes in the programming model. Similarly, the AltiVec-enabled compiler will predefine the value `__VEC__` as the decimal integer `vrrnn`.

1.2 Change Log

1.2.1 Changes to Version 1.2.3

Change the overload to vec_msum(vector signed char, vector unsigned char). Add mappings to vec_st, vec_ste, and vec_stl for vector bool types and vector pixel types. Parameters may be any integral type where type `int` was previously specified. Casting may require a parenthesized expression. The bit numbering of VRsave is reversed. The programming interface section is added. Clarify the semantics of the vector and pixel type specifiers.

1.2.2 Changes to Version 1.2.2

Add long * and unsigned long * mappings to vec_dst, vec_dstst, vec_dststt, vec_dstt, vec_ld, vec_lde, vec_ldl, vec_lvsl, vec_lvsr, vec_st, vec_ste, and vec_stl. Specify mappings for pointers to const and volatile qualified types. Fixed the range of values for vec_splat_u16. Fixed the descriptions of vec_splat_u{8,16,32} and vec_unpack2{u,s}{h,l} and vec_ste.

1.2.3 Changes to Version 1.2.1

The vector save/restore functions use r0 and modify r12. Used the conventional bit numbers for VRsave. Noted exceptions for the specific AltiVec operations. Added implementation notes for the AltiVec predicates.

1.2.4 Changes to Version 1.2.0

Typos. Changed the memory allocation routines. VMX is now vec or AltiVec. Specified scanf behavior. Described alternatives for handling the new keywords.

1.2.5 Changes to Version 1.1.8

Renamed and changed the vector save and restore functions. Changed the function prologue and epilogue sample code. Described the computation of len in the prologue and epilogue sample code. Corrected the description of vec_st and vec_stl.

1.2.6 Changes to Version 1.1.7

Added specification for all PowerPC ABIs.

1.2.7 Changes to Version 1.1.6

Move ABI Discussion to a separate document.

1.2.8 Changes to Version 1.1.5

Replace types `vec_xxx` by `vector xxx`. Update casts and constants.

1.2.9 Changes to Version 1.1.4

Add `vec_any_ne, vec_all_ne, vec_any_eq, vec_all_eq` capability for boolean and pixel types.

1.2.10 Changes to Version 1.1.3

Add `unpack2[su][hl]` (sections **Error! Reference source not found.** to **Error! Reference source not found.**) to allow converting two 8-bit elements to a 16-bit element or two 16-bit elements to a 32-bit element without having to do type casting.

1.2.11 Changes to Version 1.1.2

Third argument of `vec_msum, vec_msums` fixed.
`vec_st(vector float, int, vector float *)` added.

2. High-Level Language Interface

The high-level language interface for AltiVec is intended to accomplish the following goals:

1. Provide an efficient and expressive mechanism for programmers to access AltiVec functionality from programming languages such as C and C++.

   Note: Access to AltiVec functionality from Java applications is not currently addressed by this specification, but will likely be addressed through a higher level API such as "mediaLib."

2. Define a minimal set of language extensions that unambiguously describe the intent of the programmer while minimizing the impact on existing PowerPC compilers and development tools.

3. Provide a mechanism for code written to this interface to be compiled by a C++ compiler that is not AltiVec-enabled for a target that may not include the AltiVec architectural extensions. Equivalent functionality is obtained through use of functions simulating AltiVec operations and header files and class definitions mapping vector data types into conventional C++ types.

4. Define a minimal set of library extensions needed to support AltiVec.

2.1 Data Types

The AltiVec programming model introduces a set of fundamental data types, as described in Table 1.

New C/C++ type	Interpretation of contents	Values
`vector unsigned char`	16 `unsigned char`	0...255
`vector signed char`	16 `signed char`	-128...127
`vector bool char`	16 `unsigned char`	0(F), 255 (T)
`vector unsigned short`	8 `unsigned short`	0...65535
`vector unsigned short int`	8 `unsigned short`	0...65535
`vector signed short`	8 `signed short`	-32768...32767
`vector signed short int`	8 `signed short`	-32768...32767
`vector bool short`	8 `unsigned short`	0 (F), 65535 (T)
`vector bool short int`	8 `unsigned short`	0 (F), 65535 (T)
`vector unsigned long`	4 `unsigned long`	$0...2^{32} - 1$
`vector unsigned long int`	4 `unsigned long`	$0...2^{32} - 1$
`vector signed long`	4 `signed long`	$-2^{31}...2^{31}-1$
`vector signed long int`	4 `signed long`	$-2^{31}...2^{31}-1$
`vector bool long`	4 `unsigned long`	0 (F), $2^{32} - 1$ (T)
`vector bool long int`	4 `unsigned long`	0 (F), $2^{32} - 1$ (T)
`vector float`	4 `float`	IEEE-754 values
`vector pixel`	8 `unsigned short`	1/5/5/5 pixel

Table 1. Vector Data Types

In illustrations where an algorithm could apply to multiple types, ***vec_data*** represents any one of these types. Introducing fundamental types permits the compiler to provide stronger type checking and support overloaded operations on vector types.

2.2 New Keywords

The model introduces new uses for five identifiers: `vector`, `__vector`, `pixel`, `__pixel`, and `bool` as simple type specifier keywords. Among the type specifiers used in a declaration, the `vector` type specifier must occur first. As in C and C++, the remaining type specifiers may be freely intermixed in any order, possibly with other declaration specifiers. The syntax does not allow the use of a `typedef` name as a type specifier. For example, the following is not allowed

```
typedef signed short int16;
vector int16 data;
```

These new uses may conflict with their existing use in C and C++. There are two methods that may be used to deal with this conflict. An implementation of the AltiVec programming model may choose either method.

2.2.1 The keyword and predefine method

In this method, `__vector`, `__pixel`, and `bool` are added as keywords while `vector` and `pixel` are predefined macros. `bool` is already a keyword in C++. To allow its use in C as a keyword, it is treated the same as it is in C++. This means that the C language is extended to allow `bool` alone as a set of type specifiers. Typically, this type will map to `int`.

To accomodate a conflict with other uses of the identifiers `vector` and `pixel`, the user can either `#undef` or use a command line option to remove the predefines.

2.2.2 The context sensitive keyword method

In this method, `__vector` and `__pixel` are added as keywords without regard to context while the new uses of `vector`, `pixel`, and `bool` are keywords only in the context of a type. Since `vector` must be first among the type specifiers, it can be recognized as a type specifier when a type identifier is being scanned. The new uses of `pixel` and `bool` occur after `vector` has been recognized. In all other contexts, `vector`, `pixel`, and `bool` are not reserved. This avoids conflicts such as `class vector`, `typedef int bool`, and allows the use of `vector`, `pixel`, and `bool` as identifiers for other uses.

2.3 Alignment

2.3.1 Alignment of vector types

A defined data item of any vector data type in memory is always aligned on a 16-byte boundary. A pointer to any vector data type always points to a 16-byte boundary. The compiler is responsible for aligning vector data types on 16-byte boundaries. Given that vector data is correctly aligned, a program is incorrect if it attempts to dereference a pointer to a vector type if the pointer does not contain a 16-byte aligned address. In the AltiVec architecture, an unaligned load/store does not cause an alignment exception that might lead to (slow) loading of the bytes at the given address. Instead, the low-order bits of the address are quietly ignored.

2.3.2 Alignment of non-vector types

An array of components to be loaded into vector registers need not be aligned, but will have to be accessed with attention to its alignment. Typically, this will be accomplished with the `vec_lvsr()` or `vec_lvsl()` instruction and the `vec_perm()` instruction.

2.3.3 Alignment of aggregates and unions containing vector types

Aggregates (structures and arrays) and unions containing vector types must be aligned on 16-byte boundaries and their internal organization padded, if necessary, so that each internal vector type is aligned on a 16-byte boundary. This is an extension to all ABIs (AIX, Apple Macintosh, SVR4, and EABI).

2.4 Extensions of C/C++ operators for the new types

Most C/C++ operators do not permit any of their arguments to be one of the new types. Let `a` and `b` be vector types and `p` be a pointer to a vector type. The normal C/C++ operators are extended to include the following operations.

2.4.1 sizeof()

`sizeof(a)` and `sizeof(*p)` return 16.

2.4.2 Assignment

If either the left-hand side or right hand side of an expression has a vector type, then both sides of the expression must be of the same vector type. Thus, the expression `a=b` is valid and represents assignment if `a` and `b` are of the same vector type (or if neither is a vector type). Otherwise, the expression is invalid and must be signaled as an error by the compiler.

2.4.3 Address Operator

The operation `&a` is valid if `a` is a vector type. The result of the operation is a pointer to `a`.

2.4.4 Pointer Arithmetic

The usual pointer arithmetic can be performed on `p`. In particular, `p+1` is a pointer to the next vector element after `p`.

2.4.5 Pointer Dereferencing

If `p` is a pointer to a vector type, `*p` implies either a 128-bit vector load from the address obtained by clearing the low order bits of `p`, equivalent to the instruction `vec_ld(0,p)`, or a 128-bit vector store to that address, equivalent to the instruction `vec_st(0,p)`. If it is desired to mark the data accessed as least-recently-used (LRU), the explicit instruction `vec_ldl(0,p)` or `vec_stl(0,p)` must be used.

Dereferencing a pointer to a non-vector type produces the standard behavior of either a load or a copy of the corresponding type.

Accessing of non-aligned memory must be carried out explicitly by a `vec_ld(int, type *)` operation, a `vec_ldl(int, type *)` operation, a `vec_st(int, type *)` operation or a `vec_stl(int, type *)` operation.

2.4.6 Type Casting

Pointers to old and new types may be cast back and forth to each other. Casting a pointer to a new type represents an (unchecked) assertion that the address is 16-byte aligned. Some new operators are provided to provide the equivalence of casts and data initialization.

Casts from one vector type to another are provided by normal C casts. These should not be needed frequently if the overloaded forms of operators are used. None of the casts performs a conversion; the bit pattern of the result is the same as the bit pattern of the argument that is cast.

- `(vector signed char) vec_data`
- `(vector signed short) vec_data`
- `(vector signed long) vec_data`
- `(vector unsigned char) vec_data`
- `(vector unsigned short) vec_data`
- `(vector unsigned long) vec_data`
- `(vector bool char) vec_data`
- `(vector bool short) vec_data`
- `(vector bool long) vec_data`
- `(vector float) vec_data`
- `(vector pixel) vec_data`

2.5 New operators

New operators are introduced to construct vector literals, adjust pointers, and allow full access to the functionality provided by the AltiVec architecture.

2.5.1 Vector Literals

Vector literals are written as casts of a parenthesized set of constant expressions. These literals may be used either in initialization statements or as constants in executable statements.

- `(vector unsigned char)(unsigned int)` represents a set of 16 unsigned 8-bit quantities which all have the value specified by the integer. The compiler generates code that either computes or loads the values into the register.
- `(vector unsigned char)(unsigned int, ..., unsigned int)` represents a set of 16 unsigned 8-bit quantities specified by the 16 integers. The compiler generates code that either computes or loads the values into the register.

- `(vector signed char)(int)` represents a set of 16 signed 8-bit quantities which all have the value specified by the integer. The compiler generates code that either computes or loads the values into the register.
- `(vector signed char)(int, ..., int)` represents a set of 16 signed 8-bit quantities specified by the 16 integers. The compiler generates code that either computes or loads the values into the register.
- `(vector unsigned short)(unsigned int)` represents a set of 8 unsigned 16-bit quantities which all have the value specified by the unsigned integer. The compiler generates code that either computes or loads the values into the register.
- `(vector unsigned short)(unsigned int, ..., unsigned int)` represents a set of 8 unsigned 16-bit quantities specified by the 8 unsigned integers. The compiler generates code that either computes or loads the values into the register.
- `(vector signed short)(int)` represents a set of 8 signed 16-bit quantities which all have the value specified by the integer. The compiler generates code that either computes or loads the values into the register.
- `(vector signed short)(int, ..., int)` represents a set of 8 signed 16-bit quantities specified by the 8 integers. The compiler generates code that either computes or loads the values into the register.
- `(vector unsigned long)(unsigned int)` represents a set of 4 unsigned 32-bit quantities which all have the value specified by the unsigned integer. The compiler generates code that either computes or loads the values into the register.
- `(vector unsigned long)(unsigned int, ..., unsigned int)` represents a set of 4 unsigned 32-bit quantities specified by the 4 unsigned integers. The compiler generates code that either computes or loads the values into the register.
- `(vector signed long)(int)` represents a set of 4 signed 32-bit quantities which all have the value specified by the integer. The compiler generates code that either computes or loads the values into the register.
- `(vector signed long)(int, ..., int)` represents a set of 4 signed 32-bit quantities specified by the 4 integers. The compiler generates code that either computes or loads the values into the register.
- `(vector float)(float)` represents a set of 4 floating-point quantities which all have the value specified by the floating-point value. The compiler generates code that either computes or loads the values into the register.
- `(vector float)(float, ..., float)` represents a set of 4 floating-point quantities which all have the value specified by the 4 floating-point values. The compiler generates code that either computes or loads the values into the register.

2.5.2 Vector literals and casts

The combination of vector casts and vector literals can complicate some parsers. An implementation is not required to support the cast to a vector type of a vector cast or vector literal when the operand of the cast is not a parenthesized expression. For example, the programmer may write

```
(vector unsigned char)((vector unsigned long)(1, 2, 3, 4))
(vector signed char)((vector unsigned short) variable)
```

The similar expressions below without the parenthesized expression may not be used in a conforming application

```
(vector unsigned char)(vector unsigned long)(1, 2, 3, 4)
(vector signed char)(vector unsigned short) variable
```

2.5.3 Value for adjusting pointers

`vec_step(vec_data)` produces at compile time the integer value representing the amount by which a pointer to a component of a vector data should be incremented to cause a pointer increment to increment by 16 bytes. For example, a `vector unsigned short` data type is considered to contain 8 unsigned 2-byte values. A pointer to unsigned 2-byte values used to stream through an array of unsigned 2-byte values by a full vector at a time should be incremented by `vec_step(vector unsigned short)` = 8.

- `vec_step(vector unsigned char) = vec_step(vector signed char) = vec_step(vector bool char) = 16`
- `vec_step(vector unsigned short) = vec_step(vector signed short) = vec_step(vector bool short) = 8`
- `vec_step(vector unsigned long) = vec_step(vector signed long) = vec_step(vector bool long) = 4`
- `vec_step(vector pixel) = 8`
- `vec_step(vector float) = 4`

2.5.4 New operators representing AltiVec operations

New operators are introduced to allow full access to the functionality provided by the AltiVec architecture. The new operators are represented in the programming language by language structures that parse like function calls. The names associated with these operations are all prefixed with “`vec_`”. The appearance of one of these forms can indicate:

- a generic AltiVec operation, like `vec_add()`
- a specific AltiVec operation, like `vec_vaddubm()`
- a predicate computed from a AltiVec operation like `vec_all_eq()`
- loading of a vector of components, as discussed in section 2.5.1 on page 11.

Each operator representing an AltiVec operation takes a list of arguments representing the input operands in the order in which they are shown in the architecture specification and returns a result (possibly void).

The programming model restricts the operand types that are permitted for each AltiVec operation, whether specific or generic. The programmer may override this constraint by explicitly casting arguments to permissible types.

For a specific operation, the operand types are used to determine whether the operation is acceptable within the programming model and to determine the type of the result. For example, vec_vaddubm(vector signed char, vector signed char) is acceptable in the programming model because that represents a reasonable way to do modular addition with signed bytes, while vec_vaddubs(vector signed char, vector signed char) and vec_vadduhm(vector signed char, vector signed char) are not acceptable. If permitted, the former operation would produce a result in which saturation treated the operands as unsigned, while the latter would produce a result in which adjacent pairs of signed bytes would be treated as signed halfwords.

For a generic operation, the operand types are used to determine whether the operation is acceptable, to select a particular operation according to the types of the arguments, and to determine the type of the result. For example, vec_add(vector signed char, vector signed char) will map onto vec_vaddubm() and return a result of type vector signed char, while vec_add(vector unsigned short, vector unsigned short) will map onto vec_vadduhm() and return a result of type vector unsigned short.

The AltiVec operations that set condition register CR6 (the compare dot instructions) are treated somewhat differently in the programming model. The programmer does not have access to specific register names. Instead of directly specifying a compare dot instruction, the programmer makes reference to a predicate which returns an integer value derived from the result of a compare dot instruction. As in C, this value may be used directly as a value (1 is true, 0 is false) or as a condition for branching. It is expected that the compiler will produce the minimum code needed to use the condition. The predicates all begin with vec_all_ or vec_any_. Either the true or false state of any bit that can be set by a compare dot instruction has a predicate. For example, vec_all_gt(x,y) tests the true value of bit 24 of the CR after executing some vcmpgt. instruction. To complete the coverage by predicates, additional predicates exercise compare dot instructions with reversed or duplicated arguments. As examples, vec_all_lt(x,y) performs a vcmpgtx.(y,x), and vec_all_nan(x) is mapped onto vcmpeqfp.(x,x). If the programmer wishes to have both the result of the compare dot instruction as returned in the vector register and the value of CR6, the programmer specifies two instructions. The compiler's job is to determine that these can be merged.

The table of AltiVec operations is listed in section 4.1 on page 31. The table of AltiVec predicates is listed in section 4.2 on page 66.

2.6 Programming Interface

This document does not prohibit or require an implementation to provide any set of include files or #pragma preprocessor commands. If an implementation chooses to require that an include file be used prior to the use of the syntax described in this document, it is suggested that the include file be named <altivec.h>. If an implementation chooses to support #pragma preprocessor commands, it is suggested that it provide __ALTIVEC__ as a predefined macro with a nonzero value. A suggested set of preprocessor commands are

```
#pragma altivec_codegen on | off
```

When you this pragma is on, the compiler may use altivec instructions. When you set this pragma off, the altivec_model pragma is also set to off.

```
#pragma altivec_model on | off
```

When you this pragma is on, the compiler accepts the syntax specified in this document. When you set this pragma on, the `altivec_model` pragma is also set to on.

```
#pragma altivec_vrsave on | off
```

When you this pragma is on, the compiler will maintain the `VRsave` register.

3. Application Binary Interface (ABI)

The AltiVec Programming Model extends the existing PowerPC ABIs. Here we specify extensions to the System V Application Binary Interface PowerPC Processor Supplement (SVR4 ABI), the PowerPC Embedded Application Binary Interface (EABI), Appendix A of The PowerPC Compiler Writer's Guide (AIX ABI), and the Apple Macintosh ABI (document unknown). The SVR4 ABI and EABI specifications define both a Big-Endian ABI and a Little-Endian ABI. This extension is independent of the endian mode.

3.1 Data Representation

The vector data types are 16-bytes long and 16-byte aligned. All ABIs are extended similarly. Aggregates (structures and arrays) and unions containing vector types must be aligned on 16-byte boundaries and their internal organization padded, if necessary, so that each internal vector type is aligned on a 16-byte boundary. The AIX ABI and Apple ABI specify a maximum alignment for aggregates and unions of 4-bytes; the EABI specifies a maximum alignment of 8-bytes. Increasing the alignment to 16-bytes creates the opportunity for padding or holes in the parameter lists involving these aggregates described in section 3.4.2 on page 25.

3.2 Register Usage Conventions

The register usage conventions for the vector register file are defined as follows:

Register	Intended use	Behavior across call sites
`v0-v1`	General use	Volatile (Caller save)
`v2-v13`	Parameters, general	Volatile (Caller save)
`v14-v19`	General	Volatile (Caller save)
`v20-v31`	General	Non-volatile (Callee save)
`vrsave`	Special, see below	Non-volatile (Callee save)

Table 2. AltiVec Registers

The VRsave special purpose register (SPR(256), named `vrsave` in assembly instructions) is used to inform the operating system which vector registers need to be saved and reloaded across context switches. Bit `n` of this register is set to 1 if vector register `vn` needs to be saved and restored across a context switch. Otherwise, the operating system may return that register with any value that does not violate security after a context switch. The most significant bit in the 32-bit word is considered to be bit 0.

The EABI does not use VRsave for any special purpose, but VRsave is a non-volatile register.

3.3 The Stack Frame

The stack pointer maintains 16-byte alignment in the SVR4 ABI and the AIX ABI and 8-byte alignment in the EABI and the Apple Macintosh ABI. It is not necessary to dynamically align the stack in either the SVR4 ABI or the AIX ABI, however, the alignment padding space is specified for both.

The additions to the stack frame are the vector register save area, the VRsave save word, and the alignment padding space to dynamically align the stack to a quadword boundary.

The following additional requirements apply to the stack frame:

- Before a function changes the value of `vrsave`, it shall save the value of `vrsave` at the time of entry to the function in the VRsave save word.
- The alignment padding space shall be either 0, 4, 8, or 12 bytes long so that the address of the vector register save area (and subsequent stack locations) are quadword aligned.
- If the code establishing the stack frame dynamically aligns the stack pointer, it shall update the stack pointer atomically with an `stwux` instruction. The code may assume the stack pointer on entry is aligned on an 8-byte boundary.
- Before a function changes the value in any non-volatile vector register, `vn`, it shall save the value in `vn` in the word in the vector register save area 16*(32-`n`) bytes before the low-addressed end of the alignment padding space.
- Local variables of a vector data type which need to be saved to memory will be placed on the stack frame on a 16-byte alignment boundary in the same stack frame region used for local variables of other types.

SP in the figures denotes the stack pointer (general purpose register `r1`) of the called function after it has executed code establishing its stack stack frame.

3.3.1 SVR4 ABI and EABI Stack Frame

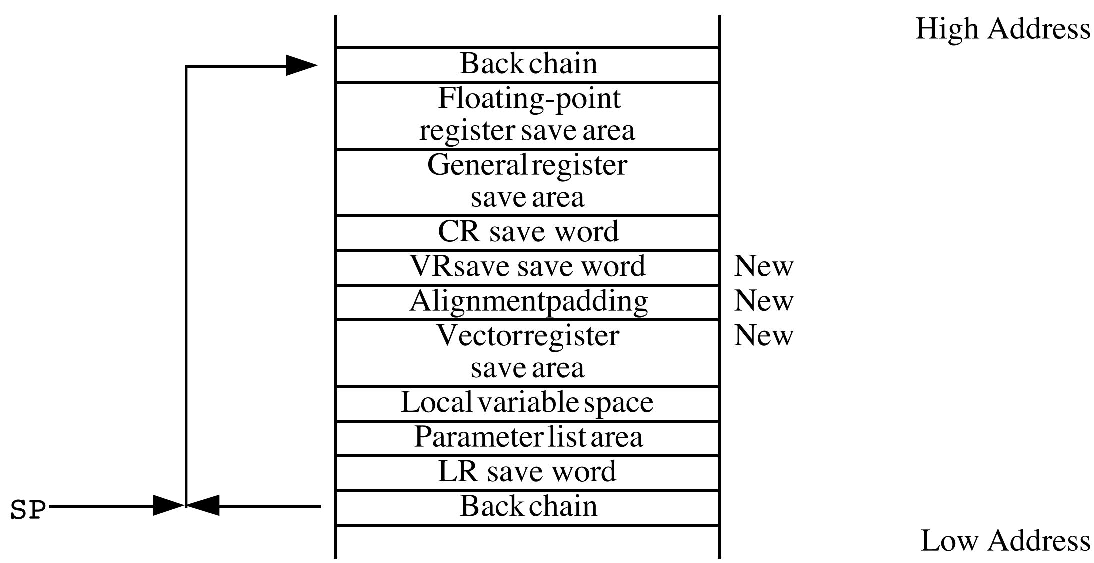

Figure 1 SVR4 ABI and EABI Stack Frame

The size of the vector register save area and the presence of the VRsave save word may vary within a function and are determined by a new Registers Valid tag. *Note: In the SVR4 ABI, the Registers Valid tag is the most general method of describing a stack frame. It is associated with a Frame or Frame Valid tag.*

Table 3. Vector Registers Valid Tag Format

Word	Bits	Name	Description
1	0-17	RESERVED	0
1	18-29	START_OFFSET	The number of words between the BASE of the nearest preceding Frame or Frame Valid tag and the first instruction to which this tag applies.
1	30-31	TYPE	2
2	0-11	VECTOR_REGS	One bit for each non-volatile vector register, bit 0 for `v31`, ..., bit 11 for `v20`, with a 1 signifying that the register is saved in the vector register save area.
2	12	VRSAVE_AREA*	1 if and only if the VRsave save word is allocated in the register save area.
2	13-17	VR *	Size in quadwords of the vector register save area.
2	18-29	RANGE	The number of words between the first and the last instruction to which this tag applies.
2	30	VRSAVE_REG	1 if and only if `vrsave` is saved in the VRsave save word.
2	31	SUBTYPE	1

* If more than one Vector Registers Valid Tag applies to the same Frame or Frame Valid tag, they shall all have the same values for VRSAVE_AREA and VR.

Figure 2 below shows sample prologue and epilogue code with full saves of all the non-volatile floating-point, general, and vector registers and a stack frame of less than 32 Kbytes. The example dynamically aligns the stack pointer, addresses incoming arguments via `r30`, uses volatile vector registers `v0-v10`, maintains `vrsave`, does not alter the non-volatile fields of the CR and does no dynamic stack allocation. Saving and restoring the vector registers and updating the `vrsave` register can occur in either order. A function that does not need to address incoming arguments but does dynamically align the stack pointer can recover the address of the original stack pointer with an instruction such as "`lwz r11,0(sp)`".

The computation of `len` in the example and whether to use `subfic` or `addi` to dynamically align the stack is based on the size of the components of the frame. Starting with the components at higher addresses, the value of `len` is computed by adding the size of the floating-point register save area, the general register save area, the CR save word, and the VRsave save word. The size of the alignment padding space is then computed as the smallest number of bytes needed to make `len` a multiple of 16. In the example below, the alignment padding space is 4 bytes. Consequently, `subfic` is used to dynamically align the stack by increasing the size of the alignment padding space by either 0 or 8 bytes. Had the alignment padding space been 8 or 12 bytes, `addi` would be used to dynamically align the stack by decreasing the size of the alignment padding space by either 0 or 8 bytes. Continuing, the value of `len` is updated by adding the size of the vector register save area, the local variable space, the outgoing parameter list area, and the LR save word. The size of the local variable space is adjusted so that the overall value of `len` is a multiple of 16.

```
function: mflr    r0              # Save return address ...
          stw     r0,4(sp)        # ... in caller's frame.
          ori     r11,sp,0        # Save end of fpr save area
          rlwinm  r12,sp,0,28,28  # 0 or 8 based on SP alignment
          subfic  r12,r12,-len    # Add in stack length
          stwux   sp,sp,r12       # Establish new aligned frame
          bl      _savefpr_14     # Save floating-point registers
          addi    r11,r11,-144    # Compute end of gpr save area
          bl      _savegpr_14_g   # Save gprs and fetch GOT ptr
          mflr    r31             # Place GOT ptr in r31
                                  # Save CR here if necessary
          addi    r30,r11,144     # Save pointer to incoming arguments
          mfspr   r0,vrsave       # Save VRsave ...
          stw     r0,-220(r30)    # ... in caller's frame.
          oris    r0,r0,0xff70    # Use v0-v10 and ...
          ori     r0,r0,0x0fff    # v20-v31 (for example)
          mtspr   vrsave,r0       # Update VRsave
          addi    r0,sp,len-224   # Compute end of vr save area
          bl      _savevr20       # Save vector registers
                                  # Body of function
          addi    r0,sp,len-224   # Address of vr save area to r0
          bl      _restvr20       # Restore vector registers
          lwz     r0,-220(r30)    # Fetch prior value of VRsave
          mtspr   vrsave,r0       # Restore VRsave
          addi    r11,r30,-144    # Address of gpr save area to r11
          bl      _restgpr_14     # Restore gprs
          addi    r11,r11,144     # Address of fpr save area to r11
          bl      _restfpr_14_x   # Restore fprs and return
```

Figure 2 SVR4 ABI and EABI Prologue and Epilogue Sample Code

3.3.2 AIX ABI and Apple Macintosh ABI Stack Frame

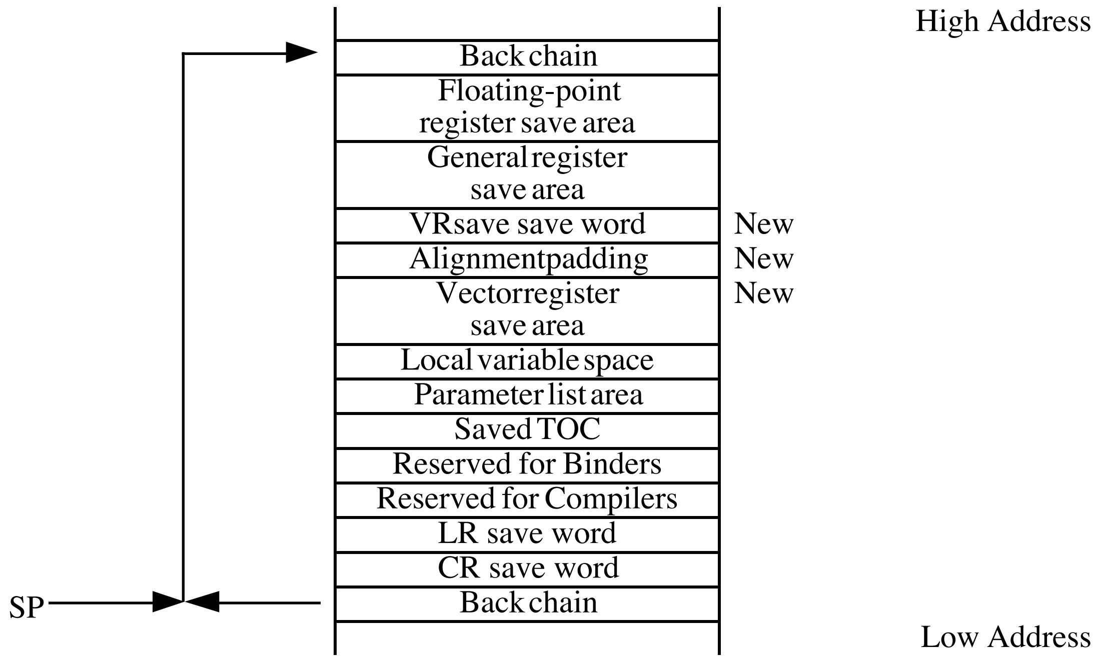

Figure 3 AIX ABI and Apple Macintosh ABI Stack Frame

The AIX ABI and Apple Macintosh ABI stack frame allows the use of a 220-byte area at a negative offset from the stack pointer. This area can be used to save non-volatile registers before the stack pointer has been updated. This size of this area is not changed. Depending on the number of non-volatile registers saved, it may be necessary to update the stack pointer before saving the vector registers. However, it remains unnecessary to update the stack pointer before saving the general-purpose registers or floating-point registers.

The size of the vector register save area and the presence of the VRsave save word are determined by a traceback table entry. The **spare3** two-bit field in the fixed portion of the traceback table is changed to:

has_vec_info One-bit field. This field is set to 1 if the procedure saves non-volatile vector registers in the vector register save area, saves `vrsave` in the VRsave save word, specifies the number of vector parameters, or uses AltiVec instructions.

spare4 One-bit field. Reserved.

When the **has_vec_info** bit is set to 1, all the following optional fields of the traceback table are present following the position of the **alloca_reg** field.

vr_saved Six-bit field. This six-bit field represents the number of non-volatile vector registers saved by this procedure. Because the last register saved is always `v31`, a value of 2 in **vr_saved** indicates that `v30` and `v31` are saved.

saves_vrsave One-bit field. If this routine saves `vrsave`, this field is set to 1. If so, the VRsave save word in the register save area must be used to restore the prior value before returning from this procedure.

has_varargs One-bit field. If this function has a variable argument list, this field is set to 1. Otherwise, it is set to 0.

vectorparms Seven-bit field. This field records the number of vector parameters. This field may be set to a non-zero value for a procedure with vector parameters that does not have a variable argument list. Otherwise, **parmsonstk** must be set.

vec_present One-bit field. This field is set to 1 if AltiVec instructions are performed within this procedure.

Note: In version 1.1.5 of this document, the vector register save area was placed between the back chain and the floating-point register save area. The new location for the vector register save area has the following advantages:

1. *The change required in the traceback table and tags mechanisms are simplified because the placement of existing elements in the stack frame does not change.*

2. *Existing debuggers and runtime exception mechanisms can support frames that contain a vector register save area by ignoring it.*

3. *The relative location of the general register save area and the floating-point register save area to the back chain is fixed.*

4. *A prologue that dynamically aligns the stack pointer may require two base registers allocated in callee-save general pupose registers. In the AIX ABI and Apple Macintosh ABI , the size of the vector register save area does not move the general register save area beyond the 220 byte area that can be saved before updating the stack pointer, thus simplifying the prologue and epilogue code.*

The prior location for the vector register save area has the following disadvantage:

1. *The alignment padding space is required when the stack pointer is 16-byte aligned and the local variable space does not need to be 16-byte aligned.*

Figure 4 below shows sample prologue and epilogue code with full saves of all the non-volatile floating-point, general, and vector registers and a stack frame of less than 32 Kbytes. The example dynamically aligns the stack pointer, addresses incoming arguments via `r31`, uses volatile vector registers `v0-v10`, maintains `vrsave`, does not alter the non-volatile fields of the CR and does no dynamic stack allocation. Saving and restoring the vector registers and updating the `vrsave` register can occur in either order. A function that does not need to address incoming arguments but does dynamically align the stack pointer can recover the address of the original stack pointer with an instruction such as "`lwz r11,0(sp)`".

The computation of `len` in the example and whether to use `subfic` or `addi` to dynamically align the stack is based on the size of the components of the frame. Starting with the components at higher addresses, the value of `len` is computed by adding the size of the floating-point register save area, the general register save area, and the VRsave save word. The size of the alignment padding space is then computed as the smallest number of bytes needed to make `len` a multiple of 16. In the example below, the alignment padding space is 0 bytes. Consequently, `subfic` is used to dynamically align the stack by increasing the size of the alignment padding space by either 0 or 8 bytes. Had the alignment padding space been 8 or 12 bytes, `addi` would be used to dynamically align the stack by decreasing the size of the alignment padding space by either 0 or 8 bytes. Continuing, the value of `len` is updated by adding the size of the vector register save area, the local variable space, the outgoing parameter list area, and 24 for the size of the link area. The size of the local varaible space is adjusted so that the overall value of `len` is a multiple of 16.

```
function: mflr    r0              # Save return address ...
          stw     r0,8(sp)        # ... in the caller's frame.
          bl      _savef14        # Save floating-point registers.
          stmw    r13,-220(sp)    # Save gprs in gpr save area
                                  # Save CR here if necessary
          ori     r31,sp,0        # Save pointer to incoming arguments
          rlwinm  r12,sp,0,28,28  # 0 or 8 based on SP alignment
          subfic  r12,r12,-len    # Add in stack length
          stwux   sp,sp,r12       # Establish new aligned frame
          mfspr   r0,vrsave       # Save VRsave ...
          stw     r0,-224(r31)    # ... in caller's frame.
          oris    r0,r0,0xff70    # Use v0-v10 v20-v31 and ...
          ori     r0,r0,0x0fff    # v20-v31 (for example)
          mtspr   vrsave,r0       # Update VRsave
          addi    r0,sp,len-224   # Compute end of vr save area
          bl      _savev20        # Save vector registers
                                  # Body of function
          addi    r0,sp,len-224   # Address of vr save area to r0
          bl      _restv20        # Restore vector registers
          lwz     r0,-224(r31)    # Fetch prior value of VRsave
          mtspr   vrsave,r0       # Restore Vrsave
          ori     sp,r31          # Restore SP
          lmw     r13,-220(sp)    # Restore gprs
          lwz     r0,8(sp)        # Restore return address ...
          mtlr    r0              # ... and return from _restf14
          b       _restf14        # Restore fprs and return
```

Figure 4 AIX ABI and Apple Macintosh ABI Prologue and Epilogue Sample Code

3.3.3 Vector Register Saving and Restoring Functions

The vector register saving and restoring functions described in this section are not part of the ABI. They are defined here only to encourage uniformity among compilers in the code used to save and restore vector registers.

On entry to the functions described in this section, `r0` contains the address of the word just beyond the end of the vector register save area, and they leave `r0` undisturbed. They modify the value of `r12`.

```
_savev20:  addi  r12,r0,-192
           stvx  v20,r12,r0      # save v20
_savev21:  addi  r12,r0,-176
           stvx  v21,r12,r0      # save v21
_savev22:  addi  r12,r0,-160
           stvx  v22,r12,r0      # save v22
_savev23:  addi  r12,r0,-144
           stvx  v23,r12,r0      # save v23
_savev24:  addi  r12,r0,-128
           stvx  v24,r12,r0      # save v24
_savev25:  addi  r12,r0,-112
           stvx  v25,r12,r0      # save v25
_savev26:  addi  r12,r0,-96
           stvx  v26,r12,r0      # save v26
_savev27:  addi  r12,r0,-80
           stvx  v27,r12,r0      # save v27
_savev28:  addi  r12,r0,-64
           stvx  v28,r12,r0      # save v28
_savev29:  addi  r12,r0,-48
           stvx  v29,r12,r0      # save v29
_savev30:  addi  r12,r0,-32
           stvx  v30,r12,r0      # save v30
_savev31:  addi  r12,r0,-16
           stvx  v31,r12,r0      # save v31
           blr                   # return to prologue
```

Figure 5 Vector Register Save

```
_restv20:  addi  r12,r0,-192
           lvx   v20,r12,r0      # restore v20
_restv21:  addi  r12,r0,-176
           lvx   v21,r12,r0      # restore v21
_restv22:  addi  r12,r0,-160
           lvx   v22,r12,r0      # restore v22
_restv23:  addi  r12,r0,-144
           lvx   v23,r12,r0      # restore v23
_restv24:  addi  r12,r0,-128
           lvx   v24,r12,r0      # restore v24
_restv25:  addi  r12,r0,-112
           lvx   v25,r12,r0      # restore v25
_restv26:  addi  r12,r0,-96
           lvx   v26,r12,r0      # restore v26
_restv27:  addi  r12,r0,-80
           lvx   v27,r12,r0      # restore v27
_restv28:  addi  r12,r0,-64
           lvx   v28,r12,r0      # restore v28
_restv29:  addi  r12,r0,-48
           lvx   v29,r12,r0      # restore v29
_restv30:  addi  r12,r0,-32
           lvx   v30,r12,r0      # restore v30
_restv31:  addi  r12,r0,-16
           lvx   v31,r12,r0      # restore v31
           blr                   # return to prologue
```

Figure 6 Vector Register Restore

3.4 Function Calls

This section applies to all user functions. The AltiVec intrinsic operations are not treated as function calls, so these comments don't apply to those operations.

The first twelve vector parameters are placed in vector registers `v2` through `v13`. If fewer (or no) vector type arguments are passed, the unneeded registers are not loaded and will contain undefined values on entry to the called function.

Functions that declare a vector data type as a return value will place that return value in register `v2`.

Any function that returns a vector type or has a vector parameter requires a prototype. This requirement enables the compiler to avoid shadowing vector registers in GPRs.

3.4.1 SVR4 ABI and EABI Parameter Passing and Varargs

The SVR4 ABI algorithm for passing parameters considers the arguments as ordered from left (first argument) to right, although the order of evaluation of the arguments is unspecified. The vector arguments maintain their ordering. The algorithm is modified to add `vr` to contain the number of the next available vector register. In the INITIALIZE step, set `vr=2`. In the SCAN loop, add a case for the next argument VECTOR_ARG as:

VECTOR_ARG:

If the next argument is in the variable portion of a parameter list, set `vr=14`. This leaves the fixed portion of a variable argument list in vector registers and places the variable portion in memory.

If `vr>13` (that is, there are no more available vector registers), go to OTHER. Otherwise, load the argument value into vector register `vr`, set `vr` to `vr+1`, and go to SCAN.

The OTHER case is modified only to understand that vector arguments have 16-byte size and alignment.

Aggregates are passed by reference (i.e., converted to a pointer to the object), so no change is needed to deal with 16-byte aligned aggregates.

The `va_list` type is unchanged, but an additional `_va_arg_type` value of 4 named `arg_VECTOR` is defined for the `__va_arg()` interface. Since vector parameters in the variable portion of a parameter list are passed in memory, the `__va_arg()` routine can access the vector value from the `overflow_arg_area` value in the `va_list` type.

3.4.2 AIX ABI and Apple Macintosh ABI Parameter Passing Without Varargs

If the function does not take a variable argument list, the non-vector parameters are passed in the same registers and stack locations as they would be if the vector parameters were not present. The only change is that aggregates and unions may be 16-byte aligned instead of 4-byte aligned. This can result in words in the parameter list being skipped for alignment (padding) and left with undefined value.

The first twelve vector parameters are placed in vector registers `v2` through `v13`. These parameters are not shadowed in GPRs. They are not allocated space in the memory argument list. Any additional vector parameters are passed through memory on the program stack. They appear together, 16-byte aligned, and after any non-vector parameters.

3.4.3 AIX ABI and Apple Macintosh ABI Parameter Passing With Varargs

The `va_list` type continues to be a pointer to the memory location of the next parameter. If `va_arg` accesses a vector type, the `va_list` value must first be aligned to a 16-byte boundary.

A function that takes a variable argument list has all parameters, including vector parameters, mapped in the argument area as ordered and aligned according to their type. The first 8 words of the argument area are shadowed in the GPRs, but only if they correspond to the variable portion of the parameter list. The first parameter word is named PW0 and is at stack offset 24. A vector parameter must be aligned on a 16-byte boundary. This means there are two cases where vector parameters are passed in GPRs. If a vector parameter is passed in PW2:PW5 (stack offset 32), its value is placed in GPRs `r5:r8`. If a vector parameter is passed in PW6:PW9 (stack offset 48), its value PW6:PW7 is placed in GPRs `r9` and `r10` and the value PW8:PW9 is placed on the stack. All parameters after the first 8 words of the argument area that correspond to the variable portion of the parameter list are passed in memory.

In the fixed portion of the parameter list, vector parameters are placed in vector registers `v2` through `v13`, but are provided a stack location corresponding to their position in the parameter list.

3.5 malloc(), vec_malloc(), and new

In the interest of saving space, malloc(), calloc(), and realloc() are not required to return a 16-byte aligned address. Instead, a new set of memory management functions is introduced that return a 16-byte aligned address. The new functions are named vec_malloc(), vec_calloc(), vec_realloc(), and vec_free(). The two sets of memory management functions may not be interchanged: memory allocated with malloc(), calloc(), or realloc() may only be freed with free() and reallocated with realloc(); memory allocated with vec_alloc(), vec_calloc(), or vec_realloc() may only be freed with vec_free() and reallocated with vec_realloc().

The user must use the appropriate set of functions based on the alignment requirement of the type involved. In the case of the C++ operator new, the implementation of new is required to use the appropriate set of functions based on the alignment requirement of the type.

3.6 setjmp() and longjmp()

The context required to be saved and restored by setjmp(), longjmp() and related functions now includes the 12 non-volatile vector registers and vrsave. The user types sigjmp_buf and jmp_buf are extended by 48 words. One of the unused words in the existing jmp_buf is used to save vrsave.

ABI	jmp_buf size	vrsave offset	v20:v31 offset
AIX ABI	448	100	256
Apple Macintosh ABI	448	16	256
SVR4 ABI and EABI	448	248	256

Open question: The SVR4 ABI states only that the sigjmp_buf *buffer is 132 words long. It does not detail what the additional 68 words are used for. It appears these are not related to the processor, but instead to the signal mechanism. The AIX ABI uses the same structure for* jmp_buf *and* sigjmp_buf*. The Apple Macintosh ABI does not seem to have a* sigjmp_buf *type.*

There are complications with setjmp() and longjmp():

1. The user types must be enlarged. Existing applications that use these interfaces will have to be recompiled even though they make no use of the AltiVec instruction set.

2. The implementation that saves and restores the vector registers can only assume that the v20:v31 offset is aligned on a 4-byte boundary. *Note: A method where the vector registers are saved at the first aligned location in the* jmp_buf *was rejected because the user types are only 4-byte aligned and may be copied by value to a location with different alignment.*

3. The implementation that saves and restores the vector registers and vrsave uses instructions that do not exist on a non-AltiVec enabled PowerPC architecture. The method for testing whether the AltiVec instructions operate is privileged. One solution is to define an O/S interface that saves and restores the vector registers and vrsave if and only if the AltiVec instructions exist and are enabled.

3.7 Debugging Information

Extensions to the debugging information format are required to describe vector types and vector register locations. While vector types can be described as fixed length arrays of existing C types, the quality implementation will describe these as new fundamental types. Doing so allows a debugger to provide mechanisms to display vector values, assign vector values, create vector literals, etc.

This section is subject to change. It is intended to describe the extensions to the standard debugging formats: xcoff stabstrings, DWARF version 1.1.0, and DWARF version 2.0.0.
Xcoff stabstrings used in the AIX ABI and adopted by the Apple Macintosh ABI support the location of objects in GPRs and FPRs. The stabstring code "R" describes a parameter passed by value in the given GPR; "r" describes a local variable residing in the given GPR. The stabstring code "X" is taken to describe a parameter passed by value in the given vector register; "x" is taken to describe a local variable residing in the given vector register.

DWARF 2.0 debugging DIEs support the location of objects in any machine register. The SVR4 ABI specifies the DWARF register number mapping. The vector registers `v0-v31` are assigned register numbers 1124-1155. The VRsave SPR is SPR256 and is assigned the register number 356.

3.8 `printf()` and `scanf()` control strings

The conversion specifications in control strings for input functions (`fscanf`, `scanf`, `sscanf`) and output functions (`fprintf`, `printf`, `sprintf`, `vfprintf`, `vprintf`, `vsprintf`) are extended to support vector types.

3.8.1 Output conversion specifications

The output conversion specifications have the following general form:

```
%[<flags>][<width>][<precision>][<size>]<conversion>
```

where,

```
<flags>           ::=  <std-flags> |
                       <c-sep>  |
                       [<std-flags>]<num-sep>[<std-flags]
<std-flags>       ::=  <std-flags-char> | <std-flags><std-flag-char>
<std-flag-char>   ::=  '-' | '+' | '0' | '#' | ' '
<c-sep>           ::=  any character except '.', '*', and those for which
                       isalnum() in the C locale is nonzero.
<num-sep>         ::=  <c-sep> except  <std-flag-char>

<width>           ::=  <decimal-integer> | '*'

<precision>       ::=  '.' <width>

<size>            ::=  'll' | 'L' | 'l' | 'h' | <vector-size>
<vector-size>     ::=  'vl' | 'vh' | 'lv' | 'hv' | 'v'
```

```
<conversion>     ::=  <char-conv> | <str-conv> | <fp-conv> |
                      <int-conv> | <misc-conv>
<char-conv>      ::=  'c'
<str-conv>       ::=  's' | 'P'
<fp-conv>        ::=  'e' | 'E' | 'f' | 'g' | 'G'
<int-conv>       ::=  'd' | 'i' | 'u' | 'o' | 'p' | 'x' | 'X'
<misc-conv>      ::=  'n' | '%'
```

The extensions to the output conversion specification for vector types are shown in **bold**.

Note: alphanumeric characters are explicitly excluded as separators.

The <vector-size> indicates that a single vector value is to be converted. The vector value is displayed in the following general form:

$value_1$ C $value_2$ C ... C $value_n$

where C is a separator character defined by the <flags> (<c-sep> or <num-sep>) and there are 4, 8, or 16 output values depending on the <vector-size> each formatted according to the <conversion>.

A <vector-size> of 'vl' or 'lv' consumes one argument and modifies the <int-conv> conversion; it should be of type vector signed long, vector unsigned long, or vector bool long; it is treated as a series of four 4-byte components. A <vector-size> of 'vh' or 'hv' consumes one argument and modifies the <int-conv> conversion; it should be of type vector signed short, vector unsigned short, vector bool short, or vector pixel; it is treated as a series of eight 2-byte components. A <vector-size> of 'v' with <int-conv> or <char-conv> consumes one argument; it should be of type vector signed char, vector unsigned char, or vector bool char; it is treated as a series of sixteen 1-byte components. A <vector-size> of 'v' with <fp-conv> consumes one argument; it should be of type vector float; it is treated as a series of four 4-byte floating-point components. All other combinations of <vector-size> and <conversion> are undefined.

The default value for the separator character is a space unless the 'c' conversion is being used. For the 'c' conversion the default separator character is null. Also for the 'c' conversion, any of the standard numeric flag characters ('-', '+', '#', ' ') may be used as a separator since these flags are not otherwise used. For numeric conversions the standard flags apply to the conversions and thus may not be specified as a separator flag. Also, only one separator character may be specified in the <flags>.

Examples:

```
vector signed char s8 = vector signed char(1, 2, 3, 4, 5, 6, 7, 8,
                                         9,10,11,12,13,14,15,16);
vector unsigned short u16 = vector unsigned short('a','b','c','d',
                                                'e','f','g','h');
vector signed long s32 = vector signed long(1, 2, 3, 99);
vector float f32 = vector float(1.1, 2.2, 3.3, 4.4);
```

```
printf("s8  = %vd\n", s8);
printf("s8  = %,vd\n", s8);
printf("u16 = %vhc\n", u16);
printf("s32 = %,2lvd\n", s32);
printf("f32 = %,5.2vf\n", f32);
```

Produces the output

```
s8  = 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
s8  =1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16
u16 = abcdefgh
s32 =  1, 2, 3,99
f32 =  1.10, 2.20, 3.30, 4.40
```

3.8.2 Input conversion specifications

The input conversion specifications have the following general form:

%[<flags>][<width>][<size>]<conversion>

where,

<flags>	::=	'*' | **<c-sep> ['*'] | ['*'] <c-sep>**
<c-sep>	::=	**any character except '.', '*', and those for which isalnum() in the C locale is nonzero.**
<width>	::=	<decimal-integer>
<size>	::=	'll' | 'L' | 'l' | 'h' | **<vector-size>**
<vector-size>	::=	**'vl' | 'vh' | 'lv' | 'hv' | 'v'**
<conversion>	::=	<char-conv> | <str-conv> | <fp-conv> | <int-conv> | <misc-conv>
<char-conv>	::=	'c'
<str-conv>	::=	's' | 'P'
<fp-conv>	::=	'e' | 'E' | 'f' | 'g' | 'G'
<int-conv>	::=	'd' | 'i' | 'u' | 'o' | 'p' | 'x' | 'X'
<misc-conv>	::=	'n' | '%'

The extensions to the input conversion specification for vector types are shown in **bold**.

Note: alphanumeric characters and '.' *are explicitly excluded as separators.*

The <vector-size> indicates that a single vector value is to be scanned and converted. The vector value to be scanned is in the following general form:

$value_1$ C $value_2$ C ... C $value_n$

where C is a separator character defined by the <flags> (<c-sep> surrounded by any number of spaces) and 4, 8, or 16 values are scanned depending on the <vector-size> each value scanned according to the <conversion>.

A `<vector-size>` of `‘vl’` or `‘lv’` consumes one argument and modifies the `<int-conv>` conversion; it should be of type `vector signed long *` or `vector unsigned long *` depending on the `<int-conv>` specification; 4 values are scanned. A `<vector-size>` of `‘vh’` or `‘hv’` consumes one argument and modifies the `<int-conv>` conversion; it should be of type `vector signed *` or `vector unsigned short *` depending on the `<int-conv>` specification; 8 values are scanned. A `<vector-size>` of `‘v’` with `<int-conv>` or `<char-conv>` consumes one argument; it should be of type `vector signed char *` or `vector unsigned char *` depending on the `<int-conv>` or `<char-conv>` specification; 16 values are scanned. A `<vector-size>` of `‘v’` with `<fp-conv>` consumes one argument; it should be of type `vector float *`; 4 floating-point values are scanned. All other combinations of `<vector-size>` and `<conversion>` are undefined.

The default value for the separator character is any number of spaces unless the `‘c’` conversion is being used. For the `‘c’` conversion the default separator character is null.

If the input stream reaches end-of-file or there is a conflict between the control string and a character read from the input stream, the input functions return `EOF` and do not assign to their vector argument. When a conflict occurs, the character causing the conflict remains unread and will be processed by the next input operation.

Examples:

```
sscanf(“1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16”, “%vd”, &s8);
sscanf(“1,2, 3 ,4 , 5,6,7,8,9,10,11,12,13,14,15,16”, “%,vd”, &s8);
sscanf(“abcdefgh”, “%vhc”, &u16);
sscanf(“1, 2, 3,99”, “%,2lvd”, &s32);
sscanf(“1.10, 2.20, 3.30, 4.40”, “%,5vf”, &f32);
```

This is equivalent to:

```
vector signed char s8 = vector signed char(1, 2, 3, 4, 5, 6, 7, 8,
                                           9,10,11,12,13,14,15,16);
vector unsigned short u16 = vector unsigned short(‘a’,’b’,’c’,’d’,
                                                  ’e’,’f’,’g’,’h’);
vector signed long s32 = vector signed long(1, 2, 3, 99);
vector float f32 = vector float(1.1, 2.2, 3.3, 4.4);
```

4. New AltiVec Operations

4.1 Generic and Specific AltiVec Operators

The first set of tables is organized alphabetically by generic operation name and defines the permitted generic and specific AltiVec operations. Each table describes a single generic AltiVec operation. Each line shows a valid set of argument types for that generic AltiVec operation, the result type for that set of argument types, and the specific AltiVec instruction generated for that set of arguments. For example, `vec_add(vector unsigned char, vector unsigned char)` maps to "`vaddubm`".

In almost all cases, it is also permissible to use a specific AltiVec operator formed by adding "`vec_`" to the name of the operation in the Maps To column with that line's set of argument types. For example, `vec_vaddubm(vector unsigned char, vector unsigned char)` has the same effect as `vec_add(vector unsigned char, vector unsigned char)`. A few cases are prohibited because that set of argument types has been chosen to produce a different result type.

Any operation that is not explicitly permitted by this table is prohibited. The desperate programmer can cast arguments, if necessary, to use operators in bizarre ways. The less desperate programmer can request an extension or modification of the programming model!

4.1.1 vec_add(arg1, arg2)

Each element of the result is the sum of the corresponding elements of **arg1** and **arg2**. The arithmetic is modular for integer types.

Result	arg1	arg2	Maps To
vector unsigned char	vector unsigned char	vector unsigned char	vaddubm
vector unsigned char	vector unsigned char	vector bool char	vaddubm
vector unsigned char	vector bool char	vector unsigned char	vaddubm
vector signed char	vector signed char	vector signed char	vaddubm
vector signed char	vector signed char	vector bool char	vaddubm
vector signed char	vector bool char	vector signed char	vaddubm
vector unsigned short	vector unsigned short	vector unsigned short	vadduhm
vector unsigned short	vector unsigned short	vector bool short	vadduhm
vector unsigned short	vector bool short	vector unsigned short	vadduhm
vector signed short	vector signed short	vector signed short	vadduhm
vector signed short	vector signed short	vector bool short	vadduhm
vector signed short	vector bool short	vector signed short	vadduhm
vector unsigned long	vector unsigned long	vector unsigned long	vadduwm
vector unsigned long	vector unsigned long	vector bool long	vadduwm
vector unsigned long	vector bool long	vector unsigned long	vadduwm
vector signed long	vector signed long	vector signed long	vadduwm
vector signed long	vector signed long	vector bool long	vadduwm
vector signed long	vector bool long	vector signed long	vadduwm
vector float	vector float	vector float	vaddfp

4.1.2 vec_addc(arg1, arg2)

Each element of the result is the carry produced by adding the corresponding elements of **arg1** and **arg2**. A carry gives a value of 1; no carry gives a value of 0.

Result	arg1	arg2	Maps To
vector unsigned long	vector unsigned long	vector unsigned long	vaddcuw

4.1.3 vec_adds(arg1, arg2)

Each element of the result is the saturated sum of the corresponding elements of **arg1** and **arg2**.

Result	arg1	arg2	Maps To
vector unsigned char	vector unsigned char	vector unsigned char	vaddubs
vector unsigned char	vector unsigned char	vector bool char	vaddubs
vector unsigned char	vector bool char	vector unsigned char	vaddubs
vector signed char	vector signed char	vector signed char	vaddsbs
vector signed char	vector signed char	vector bool char	vaddsbs
vector signed char	vector bool char	vector signed char	vaddsbs
vector unsigned short	vector unsigned short	vector unsigned short	vadduhs
vector unsigned short	vector unsigned short	vector bool short	vadduhs
vector unsigned short	vector bool short	vector unsigned short	vadduhs
vector signed short	vector signed short	vector signed short	vaddshs
vector signed short	vector signed short	vector bool short	vaddshs
vector signed short	vector bool short	vector signed short	vaddshs
vector unsigned long	vector unsigned long	vector unsigned long	vadduws
vector unsigned long	vector unsigned long	vector bool long	vadduws
vector unsigned long	vector bool long	vector unsigned long	vadduws
vector signed long	vector signed long	vector signed long	vaddsws
vector signed long	vector signed long	vector bool long	vaddsws
vector signed long	vector bool long	vector signed long	vaddsws

4.1.4 vec_and(arg1, arg2)

Each element of the result is the logical AND of the corresponding elements of **arg1** and **arg2**.

Result	arg1	arg2	Maps To
vector unsigned char	vector unsigned char	vector unsigned char	vand
vector unsigned char	vector unsigned char	vector bool char	vand
vector unsigned char	vector bool char	vector unsigned char	vand
vector signed char	vector signed char	vector signed char	vand
vector signed char	vector signed char	vector bool char	vand
vector signed char	vector bool char	vector signed char	vand
vector bool char	vector bool char	vector bool char	vand
vector unsigned short	vector unsigned short	vector unsigned short	vand
vector unsigned short	vector unsigned short	vector bool short	vand
vector unsigned short	vector bool short	vector unsigned short	vand
vector signed short	vector signed short	vector signed short	vand

vector signed short	vector signed short	vector bool short	vand
vector signed short	vector bool short	vector signed short	vand
vector bool short	vector bool short	vector bool short	vand
vector unsigned long	vector unsigned long	vector unsigned long	vand
vector unsigned long	vector unsigned long	vector bool long	vand
vector unsigned long	vector bool long	vector unsigned long	vand
vector signed long	vector signed long	vector signed long	vand
vector signed long	vector signed long	vector bool long	vand
vector signed long	vector bool long	vector signed long	vand
vector bool long	vector bool long	vector bool long	vand
vector float	vector bool long	vector float	vand
vector float	vector float	vector bool long	vand
vector float	vector float	vector float	vand

4.1.5 vec_andc(arg1, arg2)

Each element of the result is the logical AND of the corresponding element of **arg1** and the one's complement of the corresponding element of **arg2**.

Result	arg1	arg2	Maps To
vector unsigned char	vector unsigned char	vector unsigned char	vandc
vector unsigned char	vector unsigned char	vector bool char	vandc
vector unsigned char	vector bool char	vector unsigned char	vandc
vector signed char	vector signed char	vector signed char	vandc
vector signed char	vector signed char	vector bool char	vandc
vector signed char	vector bool char	vector signed char	vandc
vector bool char	vector bool char	vector bool char	vandc
vector unsigned short	vector unsigned short	vector unsigned short	vandc
vector unsigned short	vector unsigned short	vector bool short	vandc
vector unsigned short	vector bool short	vector unsigned short	vandc
vector signed short	vector signed short	vector signed short	vandc
vector signed short	vector signed short	vector bool short	vandc
vector signed short	vector bool short	vector signed short	vandc
vector bool short	vector bool short	vector bool short	vandc
vector unsigned long	vector unsigned long	vector unsigned long	vandc
vector unsigned long	vector unsigned long	vector bool long	vandc
vector unsigned long	vector bool long	vector unsigned long	vandc
vector signed long	vector signed long	vector signed long	vandc
vector signed long	vector signed long	vector bool long	vandc
vector signed long	vector bool long	vector signed long	vandc
vector bool long	vector bool long	vector bool long	vandc
vector float	vector bool long	vector float	vandc
vector float	vector float	vector bool long	vandc
vector float	vector float	vector float	vandc

4.1.6 vec_avg(arg1, arg2)

Each element of the result is the average of the corresponding elements of **arg1** and **arg2**.

Result	arg1	arg2	Maps To
vector unsigned char	vector unsigned char	vector unsigned char	vavgub
vector signed char	vector signed char	vector signed char	vavgsb
vector unsigned short	vector unsigned short	vector unsigned short	vavguh
vector signed short	vector signed short	vector signed short	vavgsh
vector unsigned long	vector unsigned long	vector unsigned long	vavguw
vector signed long	vector signed long	vector signed long	vavgsw

4.1.7 vec_ceil(arg1)

Each element of the result is the largest representable floating point integer not less than the corresponding element of **arg1**.

Result	arg1	Maps To
vector float	vector float	vrfip

4.1.8 vec_cmpb(arg1, arg2)

Each element of the result is 0 if the corresponding element of **arg1** is greater than or equal to the negative of the corresponding element of **arg2** and less than or equal to the corresponding element of **arg2**. If the corresponding element of **arg2** is not negative, each element of the result will be negative if the corresponding element of **arg1** is greater than the corresponding element of **arg2** and positive if the corresponding element of **arg1** is less than the negative of the corresponding element of **arg1**.

Result	arg1	arg2	Maps To
vector signed long	vector float	vector float	vcmpbfp

4.1.9 vec_cmpeq(arg1, arg2)

Each element of the result is TRUE if the corresponding element of **arg1** is equal to the corresponding element of **arg2**. Otherwise, it returns 0.

Result	arg1	arg2	Maps To
vector bool char	vector unsigned char	vector unsigned char	vcmpequb
vector bool char	vector signed char	vector signed char	vcmpequb
vector bool short	vector unsigned short	vector unsigned short	vcmpequh
vector bool short	vector signed short	vector signed short	vcmpequh
vector bool long	vector unsigned long	vector unsigned long	vcmpequw
vector bool long	vector signed long	vector signed long	vcmpequw
vector bool long	vector float	vector float	vcmpeqfp

4.1.10 vec_cmpge(arg1, arg2)

Each element of the result is TRUE if the corresponding element of **arg1** is greater than or equal to the corresponding element of **arg2**. Otherwise, it returns 0.

Result	arg1	arg2	Maps To
vector bool long	vector float	vector float	vcmpgefp

4.1.11 vec_cmpgt(arg1, arg2)

Each element of the result is TRUE if the corresponding element of **arg1** is greater than the corresponding element of **arg2**. Otherwise, it returns 0.

Result	arg1	arg2	Maps To
vector bool char	vector unsigned char	vector unsigned char	vcmpgtub
vector bool char	vector signed char	vector signed char	vcmpgtsb
vector bool short	vector unsigned short	vector unsigned short	vcmpgtuh
vector bool short	vector signed short	vector signed short	vcmpgtsh
vector bool long	vector unsigned long	vector unsigned long	vcmpgtuw
vector bool long	vector signed long	vector signed long	vcmpgtsw
vector bool long	vector float	vector float	vcmpgtfp

4.1.12 vec_ctf(arg1, arg2)

Each element of the result is the closest floating-point representation of the number obtained by dividing the corresponding element of **arg1** by 2 to the power of **arg2**.

Result	arg1	arg2	Maps To
vector float	vector unsigned long	5-bit unsigned literal	vcfux
vector float	vector signed long	5-bit unsigned literal	vcfsx

4.1.13 vec_cts(arg1, arg2)

Each element of the result is the saturated signed value obtained after truncating the number obtained by multiplying the corresponding element of **arg1** by 2 to the power of **arg2**.

Result	arg1	arg2	Maps To
vector signed long	vector float	5-bit unsigned literal	vctsxs

4.1.14 vec_ctu(arg1, arg2)

Each element of the result is the saturated unsigned value obtained after truncating the number obtained by multiplying the corresponding element of **arg1** by 2 to the power of **arg2**.

Result	arg1	arg2	Maps To
vector unsigned long	vector float	5-bit unsigned literal	vctuxs

4.1.15 vec_dss(arg1)

Each operation stops cache touches for the data stream associated with tag **arg1**.

Result	arg1	Maps To
void	2-bit unsigned literal	dss

4.1.16 vec_dssall(void)

The operation stops cache touches for all data streams.

Result	Maps To
void	dssall

4.1.17 vec_dst(arg1, arg2, arg3)

Each operation initiates cache touches for loads for the data stream associated with tag **arg3** at the address **arg1** using the data block in **arg2**. The **arg1** type may also be a pointer to a const-qualified type. Plain char * is excluded in the mapping for **arg1**.

Result	arg1	arg2	arg3 Maps To
void	vector unsigned char *	any integral type	2-bit unsigned literal dst
void	vector signed char *	any integral type	2-bit unsigned literal dst
void	vector bool char *	any integral type	2-bit unsigned literal dst
void	vector unsigned short *	any integral type	2-bit unsigned literal dst
void	vector signed short *	any integral type	2-bit unsigned literal dst
void	vector bool short *	any integral type	2-bit unsigned literal dst
void	vector pixel *	any integral type	2-bit unsigned literal dst
void	vector unsigned long *	any integral type	2-bit unsigned literal dst
void	vector signed long *	any integral type	2-bit unsigned literal dst
void	vector bool long *	any integral type	2-bit unsigned literal dst
void	vector float *	any integral type	2-bit unsigned literal dst
void	unsigned char *	any integral type	2-bit unsigned literal dst
void	signed char *	any integral type	2-bit unsigned literal dst
void	unsigned short *	any integral type	2-bit unsigned literal dst

void	short *	any integral type	2-bit unsigned literal dst
void	unsigned int *	any integral type	2-bit unsigned literal dst
void	int *	any integral type	2-bit unsigned literal dst
void	unsigned long *	any integral type	2-bit unsigned literal dst
void	long *	any integral type	2-bit unsigned literal dst
void	float *	any integral type	2-bit unsigned literal dst

4.1.18 vec_dstst(arg1, arg2, arg3)

Each operation initiates cache touches for stores for the data stream associated with tag **arg3** at the address **arg1** using the data block in **arg2**. The **arg1** type may also be a pointer to a const-qualified type. Plain char * is excluded in the mapping for **arg1.**

Result	arg1	arg2	arg3 Maps To
void	vector unsigned char *	any integral type	2-bit unsigned literal dstst
void	vector signed char *	any integral type	2-bit unsigned literal dstst
void	vector bool char *	any integral type	2-bit unsigned literal dstst
void	vector unsigned short *	any integral type	2-bit unsigned literal dstst
void	vector signed short *	any integral type	2-bit unsigned literal dstst
void	vector bool short *	any integral type	2-bit unsigned literal dstst
void	vector pixel *	any integral type	2-bit unsigned literal dstst
void	vector unsigned long *	any integral type	2-bit unsigned literal dstst
void	vector signed long *	any integral type	2-bit unsigned literal dstst
void	vector bool long *	any integral type	2-bit unsigned literal dstst
void	vector float *	any integral type	2-bit unsigned literal dstst
void	unsigned char *	any integral type	2-bit unsigned literal dstst
void	signed char *	any integral type	2-bit unsigned literal dstst
void	unsigned short *	any integral type	2-bit unsigned literal dstst

void	short *	any integral type	2-bit unsigned literal dstst
void	unsigned int *	any integral type	2-bit unsigned literal dstst
void	int *	any integral type	2-bit unsigned literal dstst
void	unsigned long *	any integral type	2-bit unsigned literal dstst
void	long *	any integral type	2-bit unsigned literal dstst
void	float *	any integral type	2-bit unsigned literal dstst

4.1.19 vec_dststt(arg1, arg2, arg3)

Each operation initiates cache touches for transient stores for the data stream associated with tag **arg3** at the address **arg1** using the data block in **arg2**. The **arg1** type may also be a pointer to a const-qualified type. Plain char * is excluded in the mapping for **arg1.**

Result	arg1	arg2	arg3 Maps To
void	vector unsigned char *	any integral type	2-bit unsigned literal dststt
void	vector signed char *	any integral type	2-bit unsigned literal dststt
void	vector bool char *	any integral type	2-bit unsigned literal dststt
void	vector unsigned short *	any integral type	2-bit unsigned literal dststt
void	vector signed short *	any integral type	2-bit unsigned literal dststt
void	vector bool short *	any integral type	2-bit unsigned literal dststt
void	vector pixel *	any integral type	2-bit unsigned literal dststt
void	vector unsigned long *	any integral type	2-bit unsigned literal dststt
void	vector signed long *	any integral type	2-bit unsigned literal dststt
void	vector bool long *	any integral type	2-bit unsigned literal dststt
void	vector float *	any integral type	2-bit unsigned literal dststt
void	unsigned char *	any integral type	2-bit unsigned literal dststt
void	signed char *	any integral type	2-bit unsigned literal dststt
void	unsigned short *	any integral type	2-bit unsigned literal dststt

void	short *	any integral type	2-bit unsigned literal dststt
void	unsigned int *	any integral type	2-bit unsigned literal dststt
void	int *	any integral type	2-bit unsigned literal dststt
void	unsigned long *	any integral type	2-bit unsigned literal dststt
void	long *	any integral type	2-bit unsigned literal dststt
void	float *	any integral type	2-bit unsigned literal dststt

4.1.20 vec_dstt(arg1, arg2, arg3)

Each operation initiates cache touches for transient loads for the data stream associated with tag **arg3** at the address **arg1** using the data block in **arg2**. The **arg1** type may also be a pointer to a const-qualified type. Plain char * is excluded in the mapping for **arg1**.

Result	arg1	arg2	arg3 Maps To
void	vector unsigned char *	any integral type	2-bit unsigned literal dstt
void	vector signed char *	any integral type	2-bit unsigned literal dstt
void	vector bool char *	any integral type	2-bit unsigned literal dstt
void	vector unsigned short *	any integral type	2-bit unsigned literal dstt
void	vector signed short *	any integral type	2-bit unsigned literal dstt
void	vector bool short *	any integral type	2-bit unsigned literal dstt
void	vector pixel *	any integral type	2-bit unsigned literal dstt
void	vector unsigned long *	any integral type	2-bit unsigned literal dstt
void	vector signed long *	any integral type	2-bit unsigned literal dstt
void	vector bool long *	any integral type	2-bit unsigned literal dstt
void	vector float *	any integral type	2-bit unsigned literal dstt
void	unsigned char *	any integral type	2-bit unsigned literal dstt
void	signed char *	any integral type	2-bit unsigned literal dstt
void	unsigned short *	any integral type	2-bit unsigned literal dstt

void	short *	any integral type	2-bit unsigned literal dstt
void	unsigned int *	any integral type	2-bit unsigned literal dstt
void	int *	any integral type	2-bit unsigned literal dstt
void	unsigned long *	any integral type	2-bit unsigned literal dstt
void	long *	any integral type	2-bit unsigned literal dstt
void	float *	any integral type	2-bit unsigned literal dstt

4.1.21 vec_expte(arg1)

Each element of the result is an estimate of 2 raised to the corresponding element of **arg1**.

Result	arg1	Maps To
vector float	vector float	vexptefp

4.1.22 vec_floor(arg1)

Each element of the result is the largest representable floating point integer not greater than **arg1**.

Result	arg1	Maps To
vector float	vector float	vrfim

4.1.23 vec_ld(arg1, arg2)

Each operation performs a 16-byte load at a 16-byte aligned address. **arg1** is taken to be an integer value, while **arg2** is a pointer. The sum of **arg1** and **arg2** is truncated, if necessary, to give 16-byte alignment; loading unaligned data into a vector register typically requires a permutation of the results of two loads. This load is the one that will be generated for a loading dereference of a pointer to a vector type. The **arg2** type may also be a pointer to a const-qualified type. Plain char * is excluded in the mapping for **arg2**.

Result	arg1	arg2	Maps To
vector unsigned char	any integral type	vector unsigned char *	lvx
vector unsigned char	any integral type	unsigned char *	lvx
vector signed char	any integral type	vector signed char *	lvx
vector signed char	any integral type	signed char *	lvx
vector bool char	any integral type	vector bool char *	lvx
vector unsigned short	any integral type	vector unsigned short *	lvx
vector unsigned short	any integral type	unsigned short *	lvx
vector signed short	any integral type	vector signed short *	lvx
vector signed short	any integral type	short *	lvx
vector bool short	any integral type	vector bool short *	lvx

vector pixel	any integral type	vector pixel *	lvx
vector unsigned long	any integral type	vector unsigned long *	lvx
vector unsigned long	any integral type	unsigned int *	lvx
vector unsigned long	any integral type	unsigned long *	lvx
vector signed long	any integral type	vector signed long *	lvx
vector signed long	any integral type	int *	lvx
vector signed long	any integral type	long *	lvx
vector bool long	any integral type	vector bool long *	lvx
vector float	any integral type	vector float *	lvx
vector float	any integral type	float *	lvx

4.1.24 vec_lde(arg1, arg2)

Each operation loads a single element into the position in the vector register corresponding to its address, leaving the remaining elements of the register undefined. **arg1** is taken to be an integer value, while **arg2** is a pointer. The **arg2** type may also be a pointer to a const-qualified type. Plain char * is excluded in the mapping for **arg2**.

Result	arg1	arg2	Maps To
vector unsigned char	any integral type	unsigned char *	lvebx
vector signed char	any integral type	signed char *	lvebx
vector unsigned short	any integral type	unsigned short *	lvehx
vector signed short	any integral type	short *	lvehx
vector unsigned long	any integral type	unsigned int *	lvewx
vector unsigned long	any integral type	unsigned long *	lvewx
vector signed long	any integral type	int *	lvewx
vector signed long	any integral type	long *	lvewx
vector float	any integral type	float *	lvewx

4.1.25 vec_ldl(arg1, arg2)

Each operation performs a 16-byte load at a 16-byte aligned address. **arg1** is taken to be an integer value, while **arg2** is a pointer. The sum of **arg1** and **arg2** is truncated, if necessary, to give 16-byte alignment; loading unaligned data into a vector register typically requires a permutation of the results of two loads. These operations mark the cache line as least-recently-used. The **arg2** type may also be a pointer to a const-qualified type. Plain char * is excluded in the mapping for **arg2**.

Result	arg1	arg2	Maps To
vector unsigned char	any integral type	vector unsigned char *	lvxl
vector unsigned char	any integral type	unsigned char *	lvxl
vector signed char	any integral type	vector signed char *	lvxl
vector signed char	any integral type	signed char *	lvxl
vector bool char	any integral type	vector bool char *	lvxl
vector unsigned short	any integral type	vector unsigned short *	lvxl
vector unsigned short	any integral type	unsigned short *	lvxl
vector signed short	any integral type	vector signed short *	lvxl
vector signed short	any integral type	short *	lvxl
vector bool short	any integral type	vector bool short *	lvxl
vector pixel	any integral type	vector pixel *	lvxl

vector unsigned long	any integral type	vector unsigned long *	lvxl
vector unsigned long	any integral type	unsigned int *	lvxl
vector unsigned long	any integral type	unsigned long *	lvxl
vector signed long	any integral type	vector signed long *	lvxl
vector signed long	any integral type	int *	lvxl
vector signed long	any integral type	long *	lvxl
vector bool long	any integral type	vector bool long *	lvxl
vector float	any integral type	vector float *	lvxl
vector float	any integral type	float *	lvxl

4.1.26 vec_loge(arg1)

Each element of the result is an estimate of the logarithm to base 2 of the corresponding element of **arg1**.

Result	arg1	Maps To
vector float	vector float	vlogefp

4.1.27 vec_lvsl(arg1, arg2)

Each operation generates a permutations useful for aligning data from an unaligned address. The **arg2** type may also be a pointer to a const or volatile qualified type. Plain char * is excluded in the mapping for **arg2**.

Result	arg1	arg2	Maps To
vector unsigned char	any integral type	unsigned char *	lvsl
vector unsigned char	any integral type	signed char *	lvsl
vector unsigned char	any integral type	unsigned short *	lvsl
vector unsigned char	any integral type	short *	lvsl
vector unsigned char	any integral type	unsigned int *	lvsl
vector unsigned char	any integral type	unsigned long *	lvsl
vector unsigned char	any integral type	int *	lvsl
vector unsigned char	any integral type	long *	lvsl
vector unsigned char	any integral type	float *	lvsl

4.1.28 vec_lvsr(arg1, arg2)

Each operation generates a permutations useful for aligning data from an unaligned address. The **arg2** type may also be a pointer to a const or volatile qualified type. Plain char * is excluded in the mapping for **arg2**.

Result	arg1	arg2	Maps To
vector unsigned char	any integral type	unsigned char *	lvsr
vector unsigned char	any integral type	signed char *	lvsr
vector unsigned char	any integral type	unsigned short *	lvsr
vector unsigned char	any integral type	short *	lvsr
vector unsigned char	any integral type	unsigned int *	lvsr
vector unsigned char	any integral type	unsigned long *	lvsr

vector unsigned char	any integral type	int *	lvsr
vector unsigned char	any integral type	long *	lvsr
vector unsigned char	any integral type	float *	lvsr

4.1.29 vec_madd(arg1, arg2, arg3)

Each element of the result is the sum of the corresponding element of **arg3** and the product of the corresponding elements of **arg1** and **arg2**.

Result	arg1	arg2	arg3 Maps To
vector float	vector float	vector float	vector float vmaddfp

4.1.30 vec_madds(arg1, arg2, arg3)

Each element of the result is the 16-bit saturated sum of the corresponding element of **arg3** and the high-order 17 bits of the product of the corresponding elements of **arg1** and **arg2**.

Result	arg1	arg2	arg3 Maps To
vector signed short	vector signed short	vector signed short	vector signed short vmhaddshs

4.1.31 vec_max(arg1, arg2)

Each element of the result is the larger of the corresponding elements of **arg1** and **arg2**.

Result	arg1	arg2	Maps To
vector unsigned char	vector unsigned char	vector unsigned char	vmaxub
vector unsigned char	vector unsigned char	vector bool char	vmaxub
vector unsigned char	vector bool char	vector unsigned char	vmaxub
vector signed char	vector signed char	vector signed char	vmaxsb
vector signed char	vector signed char	vector bool char	vmaxsb
vector signed char	vector bool char	vector signed char	vmaxsb
vector unsigned short	vector unsigned short	vector unsigned short	vmaxuh
vector unsigned short	vector unsigned short	vector bool short	vmaxuh
vector unsigned short	vector bool short	vector unsigned short	vmaxuh
vector signed short	vector signed short	vector signed short	vmaxsh
vector signed short	vector signed short	vector bool short	vmaxsh
vector signed short	vector bool short	vector signed short	vmaxsh
vector unsigned long	vector unsigned long	vector unsigned long	vmaxuw
vector unsigned long	vector unsigned long	vector bool long	vmaxuw
vector unsigned long	vector bool long	vector unsigned long	vmaxuw
vector signed long	vector signed long	vector signed long	vmaxsw
vector signed long	vector signed long	vector bool long	vmaxsw
vector signed long	vector bool long	vector signed long	vmaxsw
vector float	vector float	vector float	vmaxfp

4.1.32 vec_mergeh(arg1, arg2)

The even elements of the result are obtained left-to-right from the high elements of **arg1**. The odd elements of the result are obtained left-to-right from the high elements of **arg2**.

Result	arg1	arg2	Maps To
vector unsigned char	vector unsigned char	vector unsigned char	vmrghb
vector signed char	vector signed char	vector signed char	vmrghb
vector bool char	vector bool char	vector bool char	vmrghb
vector unsigned short	vector unsigned short	vector unsigned short	vmrghh
vector signed short	vector signed short	vector signed short	vmrghh
vector bool short	vector bool short	vector bool short	vmrghh
vector pixel	vector pixel	vector pixel	vmrghh
vector unsigned long	vector unsigned long	vector unsigned long	vmrghw
vector signed long	vector signed long	vector signed long	vmrghw
vector bool long	vector bool long	vector bool long	vmrghw
vector float	vector float	vector float	vmrghw

4.1.33 vec_mergel(arg1, arg2)

The even elements of the result are obtained left-to-right from the low elements of **arg1**. The odd elements of the result are obtained left-to-right from the low elements of **arg2**.

Result	arg1	arg2	Maps To
vector unsigned char	vector unsigned char	vector unsigned char	vmrglb
vector signed char	vector signed char	vector signed char	vmrglb
vector bool char	vector bool char	vector bool char	vmrglb
vector unsigned short	vector unsigned short	vector unsigned short	vmrglh
vector signed short	vector signed short	vector signed short	vmrglh
vector bool short	vector bool short	vector bool short	vmrglh
vector pixel	vector pixel	vector pixel	vmrglh
vector unsigned long	vector unsigned long	vector unsigned long	vmrglw
vector signed long	vector signed long	vector signed long	vmrglw
vector bool long	vector bool long	vector bool long	vmrglw
vector float	vector float	vector float	vmrglw

4.1.34 vec_mfvscr(void)

The first six elements of the result are 0. The seventh element of the result contains the high-order 16 bits of the VSCR (including NJ). The eighth element of the result contains the low-order 16 bits of the VSCR (including SAT).

Result	Maps To
vector unsigned short	mfvscr

4.1.35 vec_min(arg1, arg2)

Each element of the result is the smaller of the corresponding elements of **arg1** and **arg2**.

Result	arg1	arg2	Maps To
vector unsigned char	vector unsigned char	vector unsigned char	vminub
vector unsigned char	vector unsigned char	vector bool char	vminub
vector unsigned char	vector bool char	vector unsigned char	vminub
vector signed char	vector signed char	vector signed char	vminsb
vector signed char	vector signed char	vector bool char	vminsb
vector signed char	vector bool char	vector signed char	vminsb
vector unsigned short	vector unsigned short	vector unsigned short	vminuh
vector unsigned short	vector unsigned short	vector bool short	vminuh
vector unsigned short	vector bool short	vector unsigned short	vminuh
vector signed short	vector signed short	vector signed short	vminsh
vector signed short	vector signed short	vector bool short	vminsh
vector signed short	vector bool short	vector signed short	vminsh
vector unsigned long	vector unsigned long	vector unsigned long	vminuw
vector unsigned long	vector unsigned long	vector bool long	vminuw
vector unsigned long	vector bool long	vector unsigned long	vminuw
vector signed long	vector signed long	vector signed long	vminsw
vector signed long	vector signed long	vector bool long	vminsw
vector signed long	vector bool long	vector signed long	vminsw
vector float	vector float	vector float	vminfp

4.1.36 vec_mladd(arg1, arg2, arg3)

Each element of the result is the low-order 16 bits of the sum of the corresponding element of **arg3** and the product of the corresponding elements of **arg1** and **arg2**.

Result	arg1	arg2	arg3 Maps To
vector unsigned short	vector unsigned short	vector unsigned short	vector unsigned short vmladduhm
vector signed short	vector unsigned short	vector signed short	vector signed short vmladduhm
vector signed short	vector signed short	vector unsigned short	vector unsigned short vmladduhm
vector signed short	vector signed short	vector signed short	vector signed short vmladduhm

4.1.37 vec_mradds(arg1, arg2, arg3)

Each element of the result is the 16-bit saturated sum of the corresponding element of **arg3** and the high-order 17 bits of the rounded product of the corresponding elements of **arg1** and **arg2**. Note that **arg2** is unsigned, while **arg1** is signed for the variant which maps to vmsumbm.

Result	arg1	arg2	arg3 Maps To
vector signed short	vector signed short	vector signed short	vector signed short vmhraddshs

4.1.38 vec_msum(arg1, arg2, arg3)

Each element of the result is the sum of the corresponding element of **arg3** and the products of the elements of **arg1** and **arg2** which overlap the positions of that element of **arg3**. The sum is performed with 32-bit modular addition.

Result	arg1	arg2	arg3 Maps To
vector unsigned long	vector unsigned char	vector unsigned char	vector unsigned long vmsumubm
vector unsigned long	vector unsigned short	vector unsigned short	vector unsigned long vmsumuhm
vector signed long	vector signed char	vector unsigned char	vector signed long vmsummbm
vector signed long	vector signed short	vector signed short	vector signed long vmsumshm

4.1.39 vec_msums(arg1, arg2, arg3)

Each element of the result is the sum of the corresponding element of **arg3** and the products of the elements of **arg1** and **arg2** which overlap the positions of that element of **arg3**. The sum is performed with 32-bit saturating addition.

Result	arg1	arg2	arg3 Maps To
vector unsigned long	vector unsigned short	vector unsigned short	vector unsigned long vmsumuhs
vector signed long	vector signed short	vector signed short	vector signed long vmsumshs

4.1.40 vec_mtvscr(arg1)

The VSCR is set by the elements in **arg1** which occupy the last 32 bits.

Result	arg1	Maps To
void	vector unsigned char	mtvscr
void	vector signed char	mtvscr
void	vector bool char	mtvscr
void	vector unsigned short	mtvscr
void	vector signed short	mtvscr
void	vector bool short	mtvscr
void	vector pixel	mtvscr
void	vector unsigned long	mtvscr
void	vector signed long	mtvscr

void	vector bool long	mtvscr

4.1.41 vec_mule(arg1, arg2)

Each element of the result is the product of the corresponding high half-width elements of **arg1** and **arg2**.

Result	arg1	arg2	Maps To
vector unsigned short	vector unsigned char	vector unsigned char	vmuleub
vector signed short	vector signed char	vector signed char	vmulesb
vector unsigned long	vector unsigned short	vector unsigned short	vmuleuh
vector signed long	vector signed short	vector signed short	vmulesh

4.1.42 vec_mulo(arg1, arg2)

Each element of the result is the product of the corresponding low half-width elements of **arg1** and **arg2**.

Result	arg1	arg2	Maps To
vector unsigned short	vector unsigned char	vector unsigned char	vmuloub
vector signed short	vector signed char	vector signed char	vmulosb
vector unsigned long	vector unsigned short	vector unsigned short	vmulouh
vector signed long	vector signed short	vector signed short	vmulosh

4.1.43 vec_nmsub(arg1, arg2, arg3)

Each element of the result is the negative of the difference of the corresponding element of **arg3** and the product of the corresponding elements of **arg1** and **arg2**.

Result	arg1	arg2	arg3 Maps To
vector float	vector float	vector float	vector float vnmsubfp

4.1.44 vec_nor(arg1, arg2)

Each element of the result is the logical NOR of the corresponding elements of **arg1** and **arg2**.

Result	arg1	arg2	Maps To
vector unsigned char	vector unsigned char	vector unsigned char	vnor
vector signed char	vector signed char	vector signed char	vnor
vector bool char	vector bool char	vector bool char	vnor
vector unsigned short	vector unsigned short	vector unsigned short	vnor
vector signed short	vector signed short	vector signed short	vnor
vector bool short	vector bool short	vector bool short	vnor
vector unsigned long	vector unsigned long	vector unsigned long	vnor
vector signed long	vector signed long	vector signed long	vnor

vector bool long	vector bool long	vector bool long	vnor
vector float	vector float	vector float	vnor

4.1.45 vec_or(arg1, arg2)

Each element of the result is the logical OR of the corresponding elements of **arg1** and **arg2**.

Result	arg1	arg2	Maps To
vector unsigned char	vector unsigned char	vector unsigned char	vor
vector unsigned char	vector unsigned char	vector bool char	vor
vector unsigned char	vector bool char	vector unsigned char	vor
vector signed char	vector signed char	vector signed char	vor
vector signed char	vector signed char	vector bool char	vor
vector signed char	vector bool char	vector signed char	vor
vector bool char	vector bool char	vector bool char	vor
vector unsigned short	vector unsigned short	vector unsigned short	vor
vector unsigned short	vector unsigned short	vector bool short	vor
vector unsigned short	vector bool short	vector unsigned short	vor
vector signed short	vector signed short	vector signed short	vor
vector signed short	vector signed short	vector bool short	vor
vector signed short	vector bool short	vector signed short	vor
vector bool short	vector bool short	vector bool short	vor
vector unsigned long	vector unsigned long	vector unsigned long	vor
vector unsigned long	vector unsigned long	vector bool long	vor
vector unsigned long	vector bool long	vector unsigned long	vor
vector signed long	vector signed long	vector signed long	vor
vector signed long	vector signed long	vector bool long	vor
vector signed long	vector bool long	vector signed long	vor
vector bool long	vector bool long	vector bool long	vor
vector float	vector bool long	vector float	vor
vector float	vector float	vector bool long	vor
vector float	vector float	vector float	vor

4.1.46 vec_pack(arg1, arg2)

Each high element of the result is the truncation of the corresponding wider element of **arg1**. Each low element of the result is the truncation of the corresponding wider element of **arg2**.

Result	arg1	arg2	Maps To
vector unsigned char	vector unsigned short	vector unsigned short	vpkuhum
vector signed char	vector signed short	vector signed short	vpkuhum
vector bool char	vector bool short	vector bool short	vpkuhum
vector unsigned short	vector unsigned long	vector unsigned long	vpkuwum
vector signed short	vector signed long	vector signed long	vpkuwum
vector bool short	vector bool long	vector bool long	vpkuwum

4.1.47 vec_packpx(arg1, arg2)

Each high element of the result is the packed pixel from the corresponding wider element of **arg1**. Each low element of the result is the packed pixel from the corresponding wider element of **arg2**.

Result	arg1	arg2	Maps To
vector pixel	vector unsigned long	vector unsigned long	vpkpx

4.1.48 vec_packs(arg1, arg2)

Each high element of the result is the saturated value of the corresponding wider element of **arg1**. Each low element of the result is the saturated value of the corresponding wider element of **arg2**.

Result	arg1	arg2	Maps To
vector unsigned char	vector unsigned short	vector unsigned short	vpkuhus
vector signed char	vector signed short	vector signed short	vpkshss
vector unsigned short	vector unsigned long	vector unsigned long	vpkuwus
vector signed short	vector signed long	vector signed long	vpkswss

4.1.49 vec_packsu(arg1, arg2)

Each high element of the result is the saturated value of the corresponding wider element of **arg1**. Each low element of the result is the saturated value of the corresponding wider element of **arg2**. The result elements are all unsigned.

Result	arg1	arg2	Maps To
vector unsigned char	vector unsigned short	vector unsigned short	vpkuhus
vector unsigned char	vector signed short	vector signed short	vpkshus
vector unsigned short	vector unsigned long	vector unsigned long	vpkuwus
vector unsigned short	vector signed long	vector signed long	vpkswus

4.1.50 vec_perm(arg1, arg2, arg3)

Each element of the result is selected independently by indexing the catenated bytes of **arg1** and **arg2** by the corresponding element of **arg3**.

Result	arg1	arg2	arg3 Maps To
vector unsigned char	vector unsigned char	vector unsigned char	vector unsigned char vperm
vector signed char	vector signed char	vector signed char	vector unsigned char vperm
vector bool char	vector bool char	vector bool char	vector unsigned char vperm
vector unsigned short	vector unsigned short	vector unsigned short	vector unsigned char vperm
vector signed short	vector signed short	vector signed short	vector unsigned char vperm
vector bool short	vector bool short	vector bool short	vector unsigned char vperm

vector pixel	vector pixel	vector pixel	vector unsigned char vperm
vector unsigned long	vector unsigned long	vector unsigned long	vector unsigned char vperm
vector signed long	vector signed long	vector signed long	vector unsigned char vperm
vector bool long	vector bool long	vector bool long	vector unsigned char vperm
vector float	vector float	vector float	vector unsigned char vperm

4.1.51 vec_re(arg1)

Each element of the result is an estimate of the reciprocal the corresponding element of **arg1**.

Result	arg1	Maps To
vector float	vector float	vrefp

4.1.52 vec_rl(arg1, arg2)

Each element of the result is the result of rotating left the corresponding element of **arg1** by the number of bits in the corresponding element of **arg2**.

Result	arg1	arg2	Maps To
vector unsigned char	vector unsigned char	vector unsigned char	vrlb
vector signed char	vector signed char	vector unsigned char	vrlb
vector unsigned short	vector unsigned short	vector unsigned short	vrlh
vector signed short	vector signed short	vector unsigned short	vrlh
vector unsigned long	vector unsigned long	vector unsigned long	vrlw
vector signed long	vector signed long	vector unsigned long	vrlw

4.1.53 vec_round(arg1)

Each element of the result is the nearest representable floating point integer to **arg1**, using IEEE round-to-nearest rounding.

Result	arg1	Maps To
vector float	vector float	vrfin

4.1.54 vec_rsqrte(arg1)

Each element of the result is an estimate of the reciprocal square root of the corresponding element of **arg1**.

Result	arg1	Maps To
vector float	vector float	vrsqrtefp

4.1.55 vec_sel(arg1, arg2, arg3)

Each bit of the result is the corresponding bit of **arg1** if the corresponding bit of **arg3** is 0. Otherwise, it is the corresponding bit of **arg2**.

Result	arg1	arg2	arg3 Maps To
vector unsigned char	vector unsigned char	vector unsigned char	vector unsigned char vsel
vector unsigned char	vector unsigned char	vector unsigned char	vector bool char vsel
vector signed char	vector signed char	vector signed char	vector unsigned char vsel
vector signed char	vector signed char	vector signed char	vector bool char vsel
vector bool char	vector bool char	vector bool char	vector unsigned char vsel
vector bool char	vector bool char	vector bool char	vector bool char vsel
vector unsigned short	vector unsigned short	vector unsigned short	vector unsigned short vsel
vector unsigned short	vector unsigned short	vector unsigned short	vector bool short vsel
vector signed short	vector signed short	vector signed short	vector unsigned short vsel
vector signed short	vector signed short	vector signed short	vector bool short vsel
vector bool short	vector bool short	vector bool short	vector unsigned short vsel
vector bool short	vector bool short	vector bool short	vector bool short vsel
vector unsigned long	vector unsigned long	vector unsigned long	vector unsigned long vsel
vector unsigned long	vector unsigned long	vector unsigned long	vector bool long vsel
vector signed long	vector signed long	vector signed long	vector unsigned long vsel
vector signed long	vector signed long	vector signed long	vector bool long vsel
vector bool long	vector bool long	vector bool long	vector unsigned long vsel
vector bool long	vector bool long	vector bool long	vector bool long vsel
vector float	vector float	vector float	vector unsigned long vsel
vector float	vector float	vector float	vector bool long vsel

4.1.56 vec_sl(arg1, arg2)

Each element of the result is the result of shifting the corresponding element of **arg1** left by the number of bits of the corresponding element of **arg2**.

Result	arg1	arg2	Maps To
vector unsigned char	vector unsigned char	vector unsigned char	vslb
vector signed char	vector signed char	vector unsigned char	vslb
vector unsigned short	vector unsigned short	vector unsigned short	vslh
vector signed short	vector signed short	vector unsigned short	vslh
vector unsigned long	vector unsigned long	vector unsigned long	vslw
vector signed long	vector signed long	vector unsigned long	vslw

4.1.57 vec_sld(arg1, arg2, arg3)

The result is obtained by selecting the top 16 bytes obtained by shifting left (unsigned) by the value of **arg3** bytes a 32-byte quantity formed by catenating **arg1** with **arg2**.

Result	arg1	arg2	arg3 Maps To
vector unsigned char	vector unsigned char	vector unsigned char	4-bit unsigned literal vsldoi
vector signed char	vector signed char	vector signed char	4-bit unsigned literal vsldoi
vector unsigned short	vector unsigned short	vector unsigned short	4-bit unsigned literal vsldoi
vector signed short	vector signed short	vector signed short	4-bit unsigned literal vsldoi
vector pixel	vector pixel	vector pixel	4-bit unsigned literal vsldoi
vector unsigned long	vector unsigned long	vector unsigned long	4-bit unsigned literal vsldoi
vector signed long	vector signed long	vector signed long	4-bit unsigned literal vsldoi
vector float	vector float	vector float	4-bit unsigned literal vsldoi

4.1.58 vec_sll(arg1, arg2)

The result is obtained by shifting **arg1** left by a number of bits specified by the last 3 bits of the last element of **arg2**.

Result	arg1	arg2	Maps To
vector unsigned char	vector unsigned char	vector unsigned char	vsl
vector unsigned char	vector unsigned char	vector unsigned short	vsl
vector unsigned char	vector unsigned char	vector unsigned long	vsl
vector signed char	vector signed char	vector unsigned char	vsl
vector signed char	vector signed char	vector unsigned short	vsl

vector signed char	vector signed char	vector unsigned long	vsl
vector bool char	vector bool char	vector unsigned char	vsl
vector bool char	vector bool char	vector unsigned short	vsl
vector bool char	vector bool char	vector unsigned long	vsl
vector unsigned short	vector unsigned short	vector unsigned char	vsl
vector unsigned short	vector unsigned short	vector unsigned short	vsl
vector unsigned short	vector unsigned short	vector unsigned long	vsl
vector signed short	vector signed short	vector unsigned char	vsl
vector signed short	vector signed short	vector unsigned short	vsl
vector signed short	vector signed short	vector unsigned long	vsl
vector bool short	vector bool short	vector unsigned char	vsl
vector bool short	vector bool short	vector unsigned short	vsl
vector bool short	vector bool short	vector unsigned long	vsl
vector pixel	vector pixel	vector unsigned char	vsl
vector pixel	vector pixel	vector unsigned short	vsl
vector pixel	vector pixel	vector unsigned long	vsl
vector unsigned long	vector unsigned long	vector unsigned char	vsl
vector unsigned long	vector unsigned long	vector unsigned short	vsl
vector unsigned long	vector unsigned long	vector unsigned long	vsl
vector signed long	vector signed long	vector unsigned char	vsl
vector signed long	vector signed long	vector unsigned short	vsl
vector signed long	vector signed long	vector unsigned long	vsl
vector bool long	vector bool long	vector unsigned char	vsl
vector bool long	vector bool long	vector unsigned short	vsl
vector bool long	vector bool long	vector unsigned long	vsl

4.1.59 vec_slo(arg1, arg2)

The result is obtained by shifting **arg1** left by a number of bytes specified by shifting the value of the last element of **arg2** by 3 bits.

Result	arg1	arg2	Maps To
vector unsigned char	vector unsigned char	vector unsigned char	vslo
vector unsigned char	vector unsigned char	vector signed char	vslo
vector signed char	vector signed char	vector unsigned char	vslo
vector signed char	vector signed char	vector signed char	vslo
vector unsigned short	vector unsigned short	vector unsigned char	vslo
vector unsigned short	vector unsigned short	vector signed char	vslo
vector signed short	vector signed short	vector unsigned char	vslo
vector signed short	vector signed short	vector signed char	vslo
vector pixel	vector pixel	vector unsigned char	vslo
vector pixel	vector pixel	vector signed char	vslo
vector unsigned long	vector unsigned long	vector unsigned char	vslo
vector unsigned long	vector unsigned long	vector signed char	vslo
vector signed long	vector signed long	vector unsigned char	vslo
vector signed long	vector signed long	vector signed char	vslo
vector float	vector float	vector unsigned char	vslo
vector float	vector float	vector signed char	vslo

4.1.60 vec_splat(arg1, arg2)

Each element of the result is component **arg2** of **arg1**.

Result	arg1	arg2	Maps To
vector unsigned char	vector unsigned char	5-bit unsigned literal	vspltb
vector signed char	vector signed char	5-bit unsigned literal	vspltb
vector bool char	vector bool char	5-bit unsigned literal	vspltb
vector unsigned short	vector unsigned short	5-bit unsigned literal	vsplth
vector signed short	vector signed short	5-bit unsigned literal	vsplth
vector bool short	vector bool short	5-bit unsigned literal	vsplth
vector pixel	vector pixel	5-bit unsigned literal	vsplth
vector unsigned long	vector unsigned long	5-bit unsigned literal	vspltw
vector signed long	vector signed long	5-bit unsigned literal	vspltw
vector bool long	vector bool long	5-bit unsigned literal	vspltw
vector float	vector float	5-bit unsigned literal	vspltw

4.1.61 vec_splat_s8(arg1)

Each element of the result is the value obtained by sign-extending **arg1**. This permits values ranging from -16 to 15 only.

Result	arg1	Maps To
vector signed char	5-bit signed literal	vspltisb

4.1.62 vec_splat_s16(arg1)

Each element of the result is the value obtained by sign-extending **arg1**. This permits values ranging from -16 to 15 only.

Result	arg1	Maps To
vector signed short	5-bit signed literal	vspltish

4.1.63 vec_splat_s32(arg1)

Each element of the result is the value obtained by sign-extending **arg1**. This permits values ranging from -16 to 15 only.

Result	arg1	Maps To
vector signed long	5-bit signed literal	vspltisw

4.1.64 vec_splat_u8(arg1)

Each element of the result is the value obtained by sign-extending **arg1** and casting it to an unsigned char value. This value will lie in the interval from 0 to 15 or in the interval from 240 to 255. Note: it is necessary to use the generic name, since the specific operation vec_vspltisb returns a vector signed char value.

Result	arg1	Maps To
vector unsigned char	5-bit signed literal	vspltisb

4.1.65 vec_splat_u16(arg1)

Each element of the result is the value obtained by sign-extending **arg1** and casting it to an unsigned short value. This value will lie in the interval from 0 to 15 and 65520 to 65535. Note: it is necessary to use the generic name, since the specific operation vec_vspltish returns a vector signed short value.

Result	arg1	Maps To
vector unsigned short	5-bit signed literal	vspltish

4.1.66 vec_splat_u32(arg1)

Each element of the result is the value obtained by sign-extending **arg1** and casting it to an unsigned long value. This value will lie in the interval from 0 to 15 and 4294967280 to 4294967295. Note: it is necessary to use the generic name, since the specific operation vec_vspltisw returns a vector signed long value.

Result	arg1	Maps To
vector unsigned long	5-bit signed literal	vspltisw

4.1.67 vec_sr(arg1, arg2)

Each element of the result is the result of shifting the corresponding element of **arg1** right by the number of bits of the corresponding element of **arg2**. Zero bits are shifted in from the left for both signed and unsigned argument types.

Result	arg1	arg2	Maps To
vector unsigned char	vector unsigned char	vector unsigned char	vsrb
vector signed char	vector signed char	vector unsigned char	vsrb
vector unsigned short	vector unsigned short	vector unsigned short	vsrh
vector signed short	vector signed short	vector unsigned short	vsrh
vector unsigned long	vector unsigned long	vector unsigned long	vsrw
vector signed long	vector signed long	vector unsigned long	vsrw

4.1.68 vec_sra(arg1, arg2)

Each element of the result is the result of shifting the corresponding element of **arg1** right by the number of bits of the corresponding element of **arg2**. Copies of the sign bit are shifted in from the left for both signed and unsigned argument types.

Result	arg1	arg2	Maps To
vector unsigned char	vector unsigned char	vector unsigned char	vsrab
vector signed char	vector signed char	vector unsigned char	vsrab
vector unsigned short	vector unsigned short	vector unsigned short	vsrah
vector signed short	vector signed short	vector unsigned short	vsrah
vector unsigned long	vector unsigned long	vector unsigned long	vsraw

vector signed long	vector signed long	vector unsigned long	vsraw

4.1.69 vec_srl(arg1, arg2)

The result is obtained by shifting **arg1** right by a number of bits specified by the last 3 bits of the last element of **arg2**.

Result	arg1	arg2	Maps To
vector unsigned char	vector unsigned char	vector unsigned char	vsr
vector unsigned char	vector unsigned char	vector unsigned short	vsr
vector unsigned char	vector unsigned char	vector unsigned long	vsr
vector signed char	vector signed char	vector unsigned char	vsr
vector signed char	vector signed char	vector unsigned short	vsr
vector signed char	vector signed char	vector unsigned long	vsr
vector bool char	vector bool char	vector unsigned char	vsr
vector bool char	vector bool char	vector unsigned short	vsr
vector bool char	vector bool char	vector unsigned long	vsr
vector unsigned short	vector unsigned short	vector unsigned char	vsr
vector unsigned short	vector unsigned short	vector unsigned short	vsr
vector unsigned short	vector unsigned short	vector unsigned long	vsr
vector signed short	vector signed short	vector unsigned char	vsr
vector signed short	vector signed short	vector unsigned short	vsr
vector signed short	vector signed short	vector unsigned long	vsr
vector bool short	vector bool short	vector unsigned char	vsr
vector bool short	vector bool short	vector unsigned short	vsr
vector bool short	vector bool short	vector unsigned long	vsr
vector pixel	vector pixel	vector unsigned char	vsr
vector pixel	vector pixel	vector unsigned short	vsr
vector pixel	vector pixel	vector unsigned long	vsr
vector unsigned long	vector unsigned long	vector unsigned char	vsr
vector unsigned long	vector unsigned long	vector unsigned short	vsr
vector unsigned long	vector unsigned long	vector unsigned long	vsr
vector signed long	vector signed long	vector unsigned char	vsr
vector signed long	vector signed long	vector unsigned short	vsr
vector signed long	vector signed long	vector unsigned long	vsr
vector bool long	vector bool long	vector unsigned char	vsr
vector bool long	vector bool long	vector unsigned short	vsr
vector bool long	vector bool long	vector unsigned long	vsr

4.1.70 vec_sro(arg1, arg2)

The result is obtained by shifting (unsigned) **arg1** right by a number of bytes specified by shifting the value of the last element of **arg2** by 3 bits.

Result	arg1	arg2	Maps To
vector unsigned char	vector unsigned char	vector unsigned char	vsro
vector unsigned char	vector unsigned char	vector signed char	vsro
vector signed char	vector signed char	vector unsigned char	vsro

vector signed char	vector signed char	vector signed char	vsro
vector unsigned short	vector unsigned short	vector unsigned char	vsro
vector unsigned short	vector unsigned short	vector signed char	vsro
vector signed short	vector signed short	vector unsigned char	vsro
vector signed short	vector signed short	vector signed char	vsro
vector pixel	vector pixel	vector unsigned char	vsro
vector pixel	vector pixel	vector signed char	vsro
vector unsigned long	vector unsigned long	vector unsigned char	vsro
vector unsigned long	vector unsigned long	vector signed char	vsro
vector signed long	vector signed long	vector unsigned char	vsro
vector signed long	vector signed long	vector signed char	vsro
vector float	vector float	vector unsigned char	vsro
vector float	vector float	vector signed char	vsro

4.1.71 vec_st(arg1, arg2, arg3)

The 16-byte value of **arg1** is stored at a 16-byte aligned address formed by truncating the last four bits of the sum of **arg2** and **arg3**. **arg2** is taken to be an integer value, while **arg3** is a pointer. This is not, by itself, an acceptable way to store aligned data to unaligned addresses. This store is the one which will be generated for a storing dereference of a pointer to a vector type. Plain char * is excluded in the mapping for **arg3**.

Result	**arg1**	**arg2**	**arg3** **Maps To**
void	vector unsigned char	any integral type	vector unsigned char * stvx
void	vector unsigned char	any integral type	unsigned char * stvx
void	vector signed char	any integral type	vector signed char * stvx
void	vector signed char	any integral type	signed char * stvx
void	vector bool char	any integral type	vector bool char * stvx
void	vector bool char	any integral type	unsigned char * stvx
void	vector bool char	any integral type	signed char * stvx
void	vector unsigned short	any integral type	vector unsigned short * stvx
void	vector unsigned short	any integral type	unsigned short * stvx
void	vector signed short	any integral type	vector signed short * stvx
void	vector signed short	any integral type	short * stvx
void	vector bool short	any integral type	vector bool short * stvx
void	vector bool short	any integral type	short * stvx

void	vector bool short	any integral type	unsigned short * stvx
void	vector pixel short	any integral type	vector pixel short * stvx
void	vector pixel short	any integral type	short * stvx
void	vector pixel short	any integral type	unsigned short * stvx
void	vector unsigned long	any integral type	vector unsigned long * stvx
void	vector unsigned long	any integral type	unsigned int * stvx
void	vector unsigned long	any integral type	unsigned long * stvx
void	vector signed long	any integral type	vector signed long * stvx
void	vector signed long	any integral type	int * stvx
void	vector signed long	any integral type	long * stvx
void	vector bool long	any integral type	vector bool long * stvx
void	vector bool long	any integral type	unsigned int * stvx
void	vector bool long	any integral type	unsigned long * stvx
void	vector bool long	any integral type	int * stvx
void	vector bool long	any integral type	long * stvx
void	vector float	any integral type	vector float * stvx
void	vector float	any integral type	float * stvx

4.1.72 vec_ste(arg1, arg2, arg3)

A single element of **arg1** is stored at the address formed by truncating the last 0 (char), 1 (short) or 2 (int, float) bits of the sum of **arg2** and **arg3**. The element stored is the one whose position in the register matches the position of the adjusted address relative to 16-byte alignment. If you don't know the alignment of the sum of **arg2** and **arg3**, you won't know which element is stored. Plain char * is excluded in the mapping for **arg3**.

Result	**arg1**	**arg2**	**arg3** **Maps To**
void	vector unsigned char	any integral type	unsigned char * stvebx
void	vector signed char	any integral type	signed char * stvebx

void	vector bool char	any integral type	unsigned char * stvebx
void	vector bool char	any integral type	signed char * stvebx
void	vector unsigned short	any integral type	unsigned short * stvehx
void	vector signed short	any integral type	short * stvehx
void	vector bool short	any integral type	unsigned short * stvehx
void	vector bool short	any integral type	short * stvehx
void	vector pixel	any integral type	unsigned short * stvehx
void	vector pixel	any integral type	short * stvehx
void	vector unsigned long	any integral type	unsigned int * stvewx
void	vector unsigned long	any integral type	unsigned long * stvewx
void	vector signed long	any integral type	int * stvewx
void	vector signed long	any integral type	long * stvewx
void	vector bool long	any integral type	unsigned int * stvewx
void	vector bool long	any integral type	unsigned long * stvewx
void	vector bool long	any integral type	int * stvewx
void	vector bool long	any integral type	long * stvewx
void	vector float	any integral type	float * stvewx

4.1.73 vec_stl(arg1, arg2, arg3)

The 16-byte value of **arg1** is stored at a 16-byte aligned address formed by truncating the last four bits of the sum of **arg2** and **arg3**. **arg2** is taken to be an integer value, while **arg3** is a pointer. This is not, by itself, an acceptable way to store aligned data to unaligned addresses. The cache line stored into is marked LRU. Plain char * is excluded in the mapping for **arg3**.

Result	arg1	arg2	arg3 Maps To
void	vector unsigned char	any integral type	vector unsigned char * stvxl
void	vector unsigned char	any integral type	unsigned char * stvxl
void	vector signed char	any integral type	vector signed char * stvxl

void	vector signed char	any integral type	signed char * stvxl
void	vector bool char	any integral type	vector bool char * stvxl
void	vector bool char	any integral type	unsigned char * stvxl
void	vector bool char	any integral type	signed char * stvxl
void	vector unsigned short	any integral type	vector unsigned short * stvxl
void	vector unsigned short	any integral type	unsigned short * stvxl
void	vector signed short	any integral type	vector signed short * stvxl
void	vector signed short	any integral type	short * stvxl
void	vector bool short	any integral type	vector bool short * stvxl
void	vector bool short	any integral type	unsigned short * stvxl
void	vector bool short	any integral type	short * stvxl
void	vector pixel	any integral type	vector pixel * stvxl
void	vector pixel	any integral type	unsigned short * stvxl
void	vector pixel	any integral type	short * stvxl
void	vector unsigned long	any integral type	vector unsigned long * stvxl
void	vector unsigned long	any integral type	unsigned int * stvxl
void	vector unsigned long	any integral type	unsigned long * stvxl
void	vector signed long	any integral type	vector signed long * stvxl
void	vector signed long	any integral type	int * stvxl
void	vector signed long	any integral type	long * stvxl
void	vector bool long	any integral type	vector bool long * stvxl
void	vector bool long	any integral type	unsigned int * stvxl
void	vector bool long	any integral type	unsigned long * stvxl
void	vector bool long	any integral type	int * stvxl
void	vector bool long	any integral type	long * stvxl

void	vector float	any integral type	vector float * stvxl
void	vector float	any integral type	float * stvxl

4.1.74 vec_sub(arg1, arg2)

Each element of the result is the difference between the corresponding elements of **arg1** and **arg2**. The arithmetic is modular for integer types.

Result	arg1	arg2	Maps To
vector unsigned char	vector unsigned char	vector unsigned char	vsububm
vector unsigned char	vector unsigned char	vector bool char	vsububm
vector unsigned char	vector bool char	vector unsigned char	vsububm
vector signed char	vector signed char	vector signed char	vsububm
vector signed char	vector signed char	vector bool char	vsububm
vector signed char	vector bool char	vector signed char	vsububm
vector unsigned short	vector unsigned short	vector unsigned short	vsubuhm
vector unsigned short	vector unsigned short	vector bool short	vsubuhm
vector unsigned short	vector bool short	vector unsigned short	vsubuhm
vector signed short	vector signed short	vector signed short	vsubuhm
vector signed short	vector signed short	vector bool short	vsubuhm
vector signed short	vector bool short	vector signed short	vsubuhm
vector unsigned long	vector unsigned long	vector unsigned long	vsubuwm
vector unsigned long	vector unsigned long	vector bool long	vsubuwm
vector unsigned long	vector bool long	vector unsigned long	vsubuwm
vector signed long	vector signed long	vector signed long	vsubuwm
vector signed long	vector signed long	vector bool long	vsubuwm
vector signed long	vector bool long	vector signed long	vsubuwm
vector float	vector float	vector float	vsubfp

4.1.75 vec_subc(arg1, arg2)

Each element of the result is the value of the carry generated by subtracting the corresponding elements of **arg1** and **arg2**. The value is 0 if a borrow occurred and 1 if no borrow occurred.

Result	arg1	arg2	Maps To
vector unsigned long	vector unsigned long	vector unsigned long	vsubcuw

4.1.76 vec_subs(arg1, arg2)

Each element of the result is the saturated difference between the corresponding elements of **arg1** and **arg2**.

Result	arg1	arg2	Maps To
vector unsigned char	vector unsigned char	vector unsigned char	vsububs
vector unsigned char	vector unsigned char	vector bool char	vsububs

vector unsigned char	vector bool char	vector unsigned char	vsububs
vector signed char	vector signed char	vector signed char	vsubsbs
vector signed char	vector signed char	vector bool char	vsubsbs
vector signed char	vector bool char	vector signed char	vsubsbs
vector unsigned short	vector unsigned short	vector unsigned short	vsubuhs
vector unsigned short	vector unsigned short	vector bool short	vsubuhs
vector unsigned short	vector bool short	vector unsigned short	vsubuhs
vector signed short	vector signed short	vector signed short	vsubshs
vector signed short	vector signed short	vector bool short	vsubshs
vector signed short	vector bool short	vector signed short	vsubshs
vector unsigned long	vector unsigned long	vector unsigned long	vsubuws
vector unsigned long	vector unsigned long	vector bool long	vsubuws
vector unsigned long	vector bool long	vector unsigned long	vsubuws
vector signed long	vector signed long	vector signed long	vsubsws
vector signed long	vector signed long	vector bool long	vsubsws
vector signed long	vector bool long	vector signed long	vsubsws

4.1.77 vec_sum4s(arg1, arg2)

Each element of the result is the 32-bit saturated sum of the corresponding element in **arg2** and all elements in **arg1** with positions overlapping those of that element.

Result	arg1	arg2	Maps To
vector unsigned long	vector unsigned char	vector unsigned long	vsum4ubs
vector signed long	vector signed char	vector signed long	vsum4sbs
vector signed long	vector signed short	vector signed long	vsum4shs

4.1.78 vec_sum2s(arg1, arg2)

The first and third elements of the result are 0. The second element of the result is the 32-bit saturated sum of the first two elements of **arg1** and the second element of **arg2**. The fourth element of the result is the 32-bit saturated sum of the last two elements of **arg1** and the fourth element of **arg2**.

Result	arg1	arg2	Maps To
vector signed long	vector signed long	vector signed long	vsum2sws

4.1.79 vec_sums(arg1, arg2)

The first three elements of the result are 0. The fourth element of the result is the 32-bit saturated sum of all elements of **arg1** and the fourth element of **arg2**.

Result	arg1	arg2	Maps To
vector signed long	vector signed long	vector signed long	vsumsws

4.1.80 vec_trunc(arg1)

Each element of the result is the value of the corresponding element of **arg1** truncated to an integral value.

Result	arg1	Maps To
vector float	vector float	vrfiz

4.1.81 vec_unpack2sh(arg1, arg2)

These operations form signed double-size elements by catenating each high element of **arg1** with the corresponding high element of **arg2**. If **arg1** is a vector of 0's, this effectively is a signed unpack of the unsigned value **arg2**. Note: it is necessary to use the generic name, since the specific operations vec_vmrghb (vec_vmrghh) with these operand types have a result type the same as the operand type.

Result	arg1	arg2	Maps To
vector signed short	vector unsigned char	vector unsigned char	vmrghb
vector signed long	vector unsigned short	vector unsigned short	vmrghh

4.1.82 vec_unpack2sl(arg1, arg2)

These operations form signed double-size elements by catenating each low element of **arg1** with the corresponding low element of **arg2**. If **arg1** is a vector of 0's, this effectively is a signed unpack of the unsigned value **arg2**. Note: it is necessary to use the generic name, since the specific operations vec_vmrglb (vec_vmrglh) with these operand types have a result type the same as the operand type.

Result	arg1	arg2	Maps To
vector signed short	vector unsigned char	vector unsigned char	vmrglb
vector signed long	vector unsigned short	vector unsigned short	vmrglh

4.1.83 vec_unpack2uh(arg1, arg2)

These operations form unsigned double-size elements by catenating each high element of **arg1** with the corresponding high element of **arg2**. If **arg1** is a vector of 0's, this effectively is an unpack of **arg2**. Note: it is necessary to use the generic name, since the specific operations vec_vmrghb (vec_vmrghh) with these operand types have a result type the same as the operand type.

Result	arg1	arg2	Maps To
vector unsigned short	vector unsigned char	vector unsigned char	vmrghb
vector unsigned long	vector unsigned short	vector unsigned short	vmrghh

4.1.84 vec_unpack2ul(arg1, arg2)

These operations form unsigned double-size elements by catenating each low element of **arg1** with the corresponding low element of **arg2**. If **arg1** is a vector of 0's, this effectively is an unpack of **arg2**. Note: it is necessary to use the generic name, since the specific operations vec_vmrglb (vec_vmrglh) with these operand types have a result type the same as the operand type.

Result	arg1	arg2	Maps To
vector unsigned short	vector unsigned char	vector unsigned char	vmrglb
vector unsigned long	vector unsigned short	vector unsigned short	vmrglh

4.1.85 vec_unpackh(arg1)

Each element of the result is the result of extending the corresponding half-width high element of **arg1**.

Result	arg1	Maps To
vector signed short	vector signed char	vupkhsb
vector bool short	vector bool char	vupkhsb
vector unsigned long	vector pixel	vupkhpx
vector signed long	vector signed short	vupkhsh
vector bool long	vector bool short	vupkhsh

4.1.86 vec_unpackl(arg1)

Each element of the result is the result of extending the corresponding half-width low element of **arg1**.

Result	arg1	Maps To
vector signed short	vector signed char	vupklsb
vector bool short	vector bool char	vupklsb
vector unsigned long	vector pixel	vupklpx
vector signed long	vector signed short	vupklsh
vector bool long	vector bool short	vupklsh

4.1.87 vec_xor(arg1, arg2)

Each element of the result is the logical XOR of the corresponding elements of **arg1** and **arg2**.

Result	arg1	arg2	Maps To
vector unsigned char	vector unsigned char	vector unsigned char	vxor
vector unsigned char	vector unsigned char	vector bool char	vxor
vector unsigned char	vector bool char	vector unsigned char	vxor
vector signed char	vector signed char	vector signed char	vxor
vector signed char	vector signed char	vector bool char	vxor
vector signed char	vector bool char	vector signed char	vxor
vector bool char	vector bool char	vector bool char	vxor
vector unsigned short	vector unsigned short	vector unsigned short	vxor
vector unsigned short	vector unsigned short	vector bool short	vxor
vector unsigned short	vector bool short	vector unsigned short	vxor
vector signed short	vector signed short	vector signed short	vxor
vector signed short	vector signed short	vector bool short	vxor
vector signed short	vector bool short	vector signed short	vxor
vector bool short	vector bool short	vector bool short	vxor
vector unsigned long	vector unsigned long	vector unsigned long	vxor
vector unsigned long	vector unsigned long	vector bool long	vxor

vector unsigned long	vector bool long	vector unsigned long	vxor
vector signed long	vector signed long	vector signed long	vxor
vector signed long	vector signed long	vector bool long	vxor
vector signed long	vector bool long	vector signed long	vxor
vector bool long	vector bool long	vector bool long	vxor
vector float	vector bool long	vector float	vxor
vector float	vector float	vector bool long	vxor
vector float	vector float	vector float	vxor

4.2 AltiVec Predicates

The second set of tables is organized alphabetically by predicate name and defines the AltiVec predicates. Each table describes a single generic predicate. Each line shows a valid set of argument types for that predicate, and the specific AltiVec instruction generated for that set of arguments. For example, `vec_any_lt(vector unsigned char, vector unsigned char)` will use the instruction "`vcmpgtb.`". The specific operations do not exist for predicates.

4.2.1 vec_all_eq(arg1, arg2)

Each predicate returns 1 if each element of **arg1** is equal to the corresponding element of **arg2**. Otherwise, it returns 0.

Result	arg1	arg2	Maps To
int	vector unsigned char	vector unsigned char	vcmpequb.
int	vector unsigned char	vector bool char	vcmpequb.
int	vector signed char	vector signed char	vcmpequb.
int	vector signed char	vector bool char	vcmpequb.
int	vector bool char	vector unsigned char	vcmpequb.
int	vector bool char	vector signed char	vcmpequb.
int	vector bool char	vector bool char	vcmpequb.
int	vector unsigned short	vector unsigned short	vcmpequh.
int	vector unsigned short	vector bool short	vcmpequh.
int	vector signed short	vector signed short	vcmpequh.
int	vector signed short	vector bool short	vcmpequh.
int	vector bool short	vector unsigned short	vcmpequh.
int	vector bool short	vector signed short	vcmpequh.
int	vector bool short	vector bool short	vcmpequh.
int	vector pixel	vector pixel	vcmpequh.
int	vector unsigned long	vector unsigned long	vcmpequw.
int	vector unsigned long	vector bool long	vcmpequw.
int	vector signed long	vector signed long	vcmpequw.
int	vector signed long	vector bool long	vcmpequw.
int	vector bool long	vector unsigned long	vcmpequw.
int	vector bool long	vector signed long	vcmpequw.
int	vector bool long	vector bool long	vcmpequw.
int	vector float	vector float	vcmpeqfp.

4.2.2 vec_all_ge(arg1, arg2)

Each predicate returns 1 if each element of **arg1** is greater than or equal to the corresponding element of **arg2**. Otherwise, it returns 0.

Result	arg1	arg2	Maps To
int	vector unsigned char	vector unsigned char	vcmpgtub.
int	vector unsigned char	vector bool char	vcmpgtub.

int	vector signed char	vector signed char	vcmpgtsb.
int	vector signed char	vector bool char	vcmpgtsb.
int	vector bool char	vector unsigned char	vcmpgtub.
int	vector bool char	vector signed char	vcmpgtsb.
int	vector unsigned short	vector unsigned short	vcmpgtuh.
int	vector unsigned short	vector bool short	vcmpgtuh.
int	vector signed short	vector signed short	vcmpgtsh.
int	vector signed short	vector bool short	vcmpgtsh.
int	vector bool short	vector unsigned short	vcmpgtuh.
int	vector bool short	vector signed short	vcmpgtsh.
int	vector unsigned long	vector unsigned long	vcmpgtuw.
int	vector unsigned long	vector bool long	vcmpgtuw.
int	vector signed long	vector signed long	vcmpgtsw.
int	vector signed long	vector bool long	vcmpgtsw.
int	vector bool long	vector unsigned long	vcmpgtuw.
int	vector bool long	vector signed long	vcmpgtsw.
int	vector float	vector float	vcmpgefp.

4.2.3 vec_all_gt(arg1, arg2)

Each predicate returns 1 if each element of **arg1** is greater than the corresponding element of **arg2**. Otherwise, it returns 0.

Result	arg1	arg2	Maps To
int	vector unsigned char	vector unsigned char	vcmpgtub.
int	vector unsigned char	vector bool char	vcmpgtub.
int	vector signed char	vector signed char	vcmpgtsb.
int	vector signed char	vector bool char	vcmpgtsb.
int	vector bool char	vector unsigned char	vcmpgtub.
int	vector bool char	vector signed char	vcmpgtsb.
int	vector unsigned short	vector unsigned short	vcmpgtuh.
int	vector unsigned short	vector bool short	vcmpgtuh.
int	vector signed short	vector signed short	vcmpgtsh.
int	vector signed short	vector bool short	vcmpgtsh.
int	vector bool short	vector unsigned short	vcmpgtuh.
int	vector bool short	vector signed short	vcmpgtsh.
int	vector unsigned long	vector unsigned long	vcmpgtuw.
int	vector unsigned long	vector bool long	vcmpgtuw.
int	vector signed long	vector signed long	vcmpgtsw.
int	vector signed long	vector bool long	vcmpgtsw.
int	vector bool long	vector unsigned long	vcmpgtuw.
int	vector bool long	vector signed long	vcmpgtsw.
int	vector float	vector float	vcmpgtfp.

4.2.4 vec_all_in(arg1, arg2)

Each predicate returns 1 if each element of **arg1** is less than or equal to the corresponding element of **arg2** and greater than or equal to the negative of the corresponding element of **arg2**. Otherwise, it returns 0.

Result	arg1	arg2	Maps To
int	vector float	vector float	vcmpbfp.

4.2.5 vec_all_le(arg1, arg2)

Each predicate returns 1 if each element of **arg1** is less than or equal to the corresponding element of **arg2**. Otherwise, it returns 0.

Result	arg1	arg2	Maps To
int	vector unsigned char	vector unsigned char	vcmpgtub.
int	vector unsigned char	vector bool char	vcmpgtub.
int	vector signed char	vector signed char	vcmpgtsb.
int	vector signed char	vector bool char	vcmpgtsb.
int	vector bool char	vector unsigned char	vcmpgtub.
int	vector bool char	vector signed char	vcmpgtsb.
int	vector unsigned short	vector unsigned short	vcmpgtuh.
int	vector unsigned short	vector bool short	vcmpgtuh.
int	vector signed short	vector signed short	vcmpgtsh.
int	vector signed short	vector bool short	vcmpgtsh.
int	vector bool short	vector unsigned short	vcmpgtuh.
int	vector bool short	vector signed short	vcmpgtsh.
int	vector unsigned long	vector unsigned long	vcmpgtuw.
int	vector unsigned long	vector bool long	vcmpgtuw.
int	vector signed long	vector signed long	vcmpgtsw.
int	vector signed long	vector bool long	vcmpgtsw.
int	vector bool long	vector unsigned long	vcmpgtuw.
int	vector bool long	vector signed long	vcmpgtsw.
int	vector float	vector float	vcmpgefp.

4.2.6 vec_all_lt(arg1, arg2)

Each predicate returns 1 if each element of **arg1** is less than the corresponding element of **arg2**. Otherwise, it returns 0.

Result	arg1	arg2	Maps To
int	vector unsigned char	vector unsigned char	vcmpgtub.
int	vector unsigned char	vector bool char	vcmpgtub.
int	vector signed char	vector signed char	vcmpgtsb.
int	vector signed char	vector bool char	vcmpgtsb.
int	vector bool char	vector unsigned char	vcmpgtub.
int	vector bool char	vector signed char	vcmpgtsb.
int	vector unsigned short	vector unsigned short	vcmpgtuh.
int	vector unsigned short	vector bool short	vcmpgtuh.

int	vector signed short	vector signed short	vcmpgtsh.
int	vector signed short	vector bool short	vcmpgtsh.
int	vector bool short	vector unsigned short	vcmpgtuh.
int	vector bool short	vector signed short	vcmpgtsh.
int	vector unsigned long	vector unsigned long	vcmpgtuw.
int	vector unsigned long	vector bool long	vcmpgtuw.
int	vector signed long	vector signed long	vcmpgtsw.
int	vector signed long	vector bool long	vcmpgtsw.
int	vector bool long	vector unsigned long	vcmpgtuw.
int	vector bool long	vector signed long	vcmpgtsw.
int	vector float	vector float	vcmpgtfp.

4.2.7 vec_all_nan(arg1)

Each predicate returns 1 if each element of **arg1** is a NaN. Otherwise, it returns 0.

Result	arg1	Maps To
int	vector float	vcmpeqfp.

4.2.8 vec_all_ne(arg1, arg2)

Each predicate returns 1 if each element of **arg1** is not equal to the corresponding element of **arg2**. Otherwise, it returns 0.

Result	arg1	arg2	Maps To
int	vector unsigned char	vector unsigned char	vcmpequb.
int	vector unsigned char	vector bool char	vcmpequb.
int	vector signed char	vector signed char	vcmpequb.
int	vector signed char	vector bool char	vcmpequb.
int	vector bool char	vector unsigned char	vcmpequb.
int	vector bool char	vector signed char	vcmpequb.
int	vector bool char	vector bool char	vcmpequb.
int	vector unsigned short	vector unsigned short	vcmpequh.
int	vector unsigned short	vector bool short	vcmpequh.
int	vector signed short	vector signed short	vcmpequh.
int	vector signed short	vector bool short	vcmpequh.
int	vector bool short	vector unsigned short	vcmpequh.
int	vector bool short	vector signed short	vcmpequh.
int	vector bool short	vector bool short	vcmpequh.
int	vector pixel	vector pixel	vcmpequh.
int	vector unsigned long	vector unsigned long	vcmpequw.
int	vector unsigned long	vector bool long	vcmpequw.
int	vector signed long	vector signed long	vcmpequw.
int	vector signed long	vector bool long	vcmpequw.
int	vector bool long	vector unsigned long	vcmpequw.
int	vector bool long	vector signed long	vcmpequw.
int	vector bool long	vector bool long	vcmpequw.
int	vector float	vector float	vcmpeqfp.

4.2.9 vec_all_nge(arg1, arg2)

Each predicate returns 1 if each element of **arg1** is not greater than or equal to the corresponding element of **arg2**. Otherwise, it returns 0. Not greater than or equal can mean either less than or that one of the elements is a NaN.

Result	arg1	arg2	Maps To
int	vector float	vector float	vcmpgefp.

4.2.10 vec_all_ngt(arg1, arg2)

Each predicate returns 1 if each element of **arg1** is not greater than the corresponding element of **arg2**. Otherwise, it returns 0. Not greater than or equal can mean either less than or equal to or that one of the elements is a NaN.

Result	arg1	arg2	Maps To
int	vector float	vector float	vcmpgtfp.

4.2.11 vec_all_nle(arg1, arg2)

Each predicate returns 1 if each element of **arg1** is not less than or equal to the corresponding element of **arg2**. Otherwise, it returns 0. Not greater than or equal can mean either greater than or that one of the elements is a NaN.

Result	arg1	arg2	Maps To
int	vector float	vector float	vcmpgefp.

4.2.12 vec_all_nlt(arg1, arg2)

Each predicate returns 1 if each element of **arg1** is not less than the corresponding element of **arg2**. Otherwise, it returns 0. Not greater than or equal can mean either greater than or equal to or that one of the elements is a NaN.

Result	arg1	arg2	Maps To
int	vector float	vector float	vcmpgtfp.

4.2.13 vec_all_numeric(arg1)

Each predicate returns 1 if each element of **arg1** is numeric (not a NaN). Otherwise, it returns 0.

Result	arg1	Maps To
int	vector float	vcmpeqfp.

4.2.14 vec_any_eq(arg1, arg2)

Each predicate returns 1 if at least one element of **arg1** is equal to the corresponding element of **arg2**. Otherwise, it returns 0.

Result	arg1	arg2	Maps To
int	vector unsigned char	vector unsigned char	vcmpequb.
int	vector unsigned char	vector bool char	vcmpequb.
int	vector signed char	vector signed char	vcmpequb.
int	vector signed char	vector bool char	vcmpequb.
int	vector bool char	vector unsigned char	vcmpequb.
int	vector bool char	vector signed char	vcmpequb.
int	vector bool char	vector bool char	vcmpequb.
int	vector unsigned short	vector unsigned short	vcmpequh.
int	vector unsigned short	vector bool short	vcmpequh.
int	vector signed short	vector signed short	vcmpequh.
int	vector signed short	vector bool short	vcmpequh.
int	vector bool short	vector unsigned short	vcmpequh.
int	vector bool short	vector signed short	vcmpequh.
int	vector bool short	vector bool short	vcmpequh.
int	vector pixel	vector pixel	vcmpequh.
int	vector unsigned long	vector unsigned long	vcmpequw.
int	vector unsigned long	vector bool long	vcmpequw.
int	vector signed long	vector signed long	vcmpequw.
int	vector signed long	vector bool long	vcmpequw.
int	vector bool long	vector unsigned long	vcmpequw.
int	vector bool long	vector signed long	vcmpequw.
int	vector bool long	vector bool long	vcmpequw.
int	vector float	vector float	vcmpeqfp.

4.2.15 vec_any_ge(arg1, arg2)

Each predicate returns 1 if at least one element of **arg1** is greater than or equal to the corresponding element of **arg2**. Otherwise, it returns 0.

Result	arg1	arg2	Maps To
int	vector unsigned char	vector unsigned char	vcmpgtub.
int	vector unsigned char	vector bool char	vcmpgtub.
int	vector signed char	vector signed char	vcmpgtsb.
int	vector signed char	vector bool char	vcmpgtsb.
int	vector bool char	vector unsigned char	vcmpgtub.
int	vector bool char	vector signed char	vcmpgtsb.
int	vector unsigned short	vector unsigned short	vcmpgtuh.
int	vector unsigned short	vector bool short	vcmpgtuh.
int	vector signed short	vector signed short	vcmpgtsh.
int	vector signed short	vector bool short	vcmpgtsh.
int	vector bool short	vector unsigned short	vcmpgtuh.
int	vector bool short	vector signed short	vcmpgtsh.
int	vector unsigned long	vector unsigned long	vcmpgtuw.

int	vector unsigned long	vector bool long	vcmpgtuw.
int	vector signed long	vector signed long	vcmpgtsw.
int	vector signed long	vector bool long	vcmpgtsw.
int	vector bool long	vector unsigned long	vcmpgtuw.
int	vector bool long	vector signed long	vcmpgtsw.
int	vector float	vector float	vcmpgefp.

4.2.16 vec_any_gt(arg1, arg2)

Each predicate returns 1 if at least one element of **arg1** is greater than the corresponding element of **arg2**. Otherwise, it returns 0.

Result	arg1	arg2	Maps To
int	vector unsigned char	vector unsigned char	vcmpgtub.
int	vector unsigned char	vector bool char	vcmpgtub.
int	vector signed char	vector signed char	vcmpgtsb.
int	vector signed char	vector bool char	vcmpgtsb.
int	vector bool char	vector unsigned char	vcmpgtub.
int	vector bool char	vector signed char	vcmpgtsb.
int	vector unsigned short	vector unsigned short	vcmpgtuh.
int	vector unsigned short	vector bool short	vcmpgtuh.
int	vector signed short	vector signed short	vcmpgtsh.
int	vector signed short	vector bool short	vcmpgtsh.
int	vector bool short	vector unsigned short	vcmpgtuh.
int	vector bool short	vector signed short	vcmpgtsh.
int	vector unsigned long	vector unsigned long	vcmpgtuw.
int	vector unsigned long	vector bool long	vcmpgtuw.
int	vector signed long	vector signed long	vcmpgtsw.
int	vector signed long	vector bool long	vcmpgtsw.
int	vector bool long	vector unsigned long	vcmpgtuw.
int	vector bool long	vector signed long	vcmpgtsw.
int	vector float	vector float	vcmpgtfp.

4.2.17 vec_any_le(arg1, arg2)

Each predicate returns 1 if at least one element of **arg1** is less than or equal to the corresponding element of **arg2**. Otherwise, it returns 0.

Result	arg1	arg2	Maps To
int	vector unsigned char	vector unsigned char	vcmpgtub.
int	vector unsigned char	vector bool char	vcmpgtub.
int	vector signed char	vector signed char	vcmpgtsb.
int	vector signed char	vector bool char	vcmpgtsb.
int	vector bool char	vector unsigned char	vcmpgtub.
int	vector bool char	vector signed char	vcmpgtsb.
int	vector unsigned short	vector unsigned short	vcmpgtuh.
int	vector unsigned short	vector bool short	vcmpgtuh.
int	vector signed short	vector signed short	vcmpgtsh.

int	vector signed short	vector bool short	vcmpgtsh.
int	vector bool short	vector unsigned short	vcmpgtuh.
int	vector bool short	vector signed short	vcmpgtsh.
int	vector unsigned long	vector unsigned long	vcmpgtuw.
int	vector unsigned long	vector bool long	vcmpgtuw.
int	vector signed long	vector signed long	vcmpgtsw.
int	vector signed long	vector bool long	vcmpgtsw.
int	vector bool long	vector unsigned long	vcmpgtuw.
int	vector bool long	vector signed long	vcmpgtsw.
int	vector float	vector float	vcmpgefp.

4.2.18 vec_any_lt(arg1, arg2)

Each predicate returns 1 if at least one element of **arg1** is less than the corresponding element of **arg2**. Otherwise, it returns 0.

Result	arg1	arg2	Maps To
int	vector unsigned char	vector unsigned char	vcmpgtub.
int	vector unsigned char	vector bool char	vcmpgtub.
int	vector signed char	vector signed char	vcmpgtsb.
int	vector signed char	vector bool char	vcmpgtsb.
int	vector bool char	vector unsigned char	vcmpgtub.
int	vector bool char	vector signed char	vcmpgtsb.
int	vector unsigned short	vector unsigned short	vcmpgtuh.
int	vector unsigned short	vector bool short	vcmpgtuh.
int	vector signed short	vector signed short	vcmpgtsh.
int	vector signed short	vector bool short	vcmpgtsh.
int	vector bool short	vector unsigned short	vcmpgtuh.
int	vector bool short	vector signed short	vcmpgtsh.
int	vector unsigned long	vector unsigned long	vcmpgtuw.
int	vector unsigned long	vector bool long	vcmpgtuw.
int	vector signed long	vector signed long	vcmpgtsw.
int	vector signed long	vector bool long	vcmpgtsw.
int	vector bool long	vector unsigned long	vcmpgtuw.
int	vector bool long	vector signed long	vcmpgtsw.
int	vector float	vector float	vcmpgtfp.

4.2.19 vec_any_nan(arg1)

Each predicate returns 1 if at least one element of **arg1** is a NaN. Otherwise, it returns 0.

Result	arg1	Maps To
int	vector float	vcmpeqfp.

4.2.20 vec_any_ne(arg1, arg2)

Each predicate returns 1 if at least one element of **arg1** is not equal to the corresponding element of **arg2**. Otherwise, it returns 0.

Result	arg1	arg2	Maps To
int	vector unsigned char	vector unsigned char	vcmpequb.
int	vector unsigned char	vector bool char	vcmpequb.
int	vector signed char	vector signed char	vcmpequb.
int	vector signed char	vector bool char	vcmpequb.
int	vector bool char	vector unsigned char	vcmpequb.
int	vector bool char	vector signed char	vcmpequb.
int	vector bool char	vector bool char	vcmpequb.
int	vector unsigned short	vector unsigned short	vcmpequh.
int	vector unsigned short	vector bool short	vcmpequh.
int	vector signed short	vector signed short	vcmpequh.
int	vector signed short	vector bool short	vcmpequh.
int	vector bool short	vector unsigned short	vcmpequh.
int	vector bool short	vector signed short	vcmpequh.
int	vector bool short	vector bool short	vcmpequh.
int	vector pixel	vector pixel	vcmpequh.
int	vector unsigned long	vector unsigned long	vcmpequw.
int	vector unsigned long	vector bool long	vcmpequw.
int	vector signed long	vector signed long	vcmpequw.
int	vector signed long	vector bool long	vcmpequw.
int	vector bool long	vector unsigned long	vcmpequw.
int	vector bool long	vector signed long	vcmpequw.
int	vector bool long	vector bool long	vcmpequw.
int	vector float	vector float	vcmpeqfp.

4.2.21 vec_any_nge(arg1, arg2)

Each predicate returns 1 if at least one element of **arg1** is not greater than or equal to the corresponding element of **arg2**. Otherwise, it returns 0. Not greater than or equal can mean either less than or that one of the elements is a NaN.

Result	arg1	arg2	Maps To
int	vector float	vector float	vcmpgefp.

4.2.22 vec_any_ngt(arg1, arg2)

Each predicate returns 1 if at least one element of **arg1** is not greater than the corresponding element of **arg2**. Otherwise, it returns 0. Not greater than can mean either less than or equal to or that one of the elements is a NaN.

Result	arg1	arg2	Maps To
int	vector float	vector float	vcmpgtfp.

4.2.23 vec_any_nle(arg1, arg2)

Each predicate returns 1 if at least one element of **arg1** is not less than or equal to the corresponding element of **arg2**. Otherwise, it returns 0. Not less than or equal can mean either greater than or that one of the elements is a NaN.

Result	arg1	arg2	Maps To
int	vector float	vector float	vcmpgefp.

4.2.24 vec_any_nlt(arg1, arg2)

Each predicate returns 1 if at least one element of **arg1** is not less than the corresponding element of **arg2**. Otherwise, it returns 0. Not less than can mean either greater than or equal to or that one of the elements is a NaN.

Result	arg1	arg2	Maps To
int	vector float	vector float	vcmpgtfp.

4.2.25 vec_any_numeric(arg1)

Each predicate returns 1 if at least one element of **arg1** is numeric (not a NaN). Otherwise, it returns 0.

Result	arg1	Maps To
int	vector float	vcmpeqfp.

4.2.26 vec_any_out(arg1, arg2)

Each predicate returns 1 if at least one element of **arg1** is not less than or equal to the corresponding element of **arg2** or not greater than or equal to the negative of the corresponding element of **arg2**. Otherwise, it returns 0. Not less than or equal can mean greater than or that either argument is a NaN. Not greater than or equal can mean less than or that either argument is a NaN.

Result	arg1	arg2	Maps To
int	vector float	vector float	vcmpbfp.

5. Future Directions

5.1 Assembly Language Interface

5.2 AltiVec Instruction Mnemonics

5.3 Compiler Implementation Notes

5.3.1 AltiVec Predicate mappings

In most cases, the predicates are implemented by supplying the operands to the instructions in the same order as the predicate arguments. All exceptions to this rule are noted below.

5.3.1.1 vec_all_ge(arg1, arg2) and vec_any_ge(arg1, arg2)

To implement the predicates for all operand types **except** for vector float, supply the operands to the instruction in reverse order.

5.3.1.2 vec_all_le(arg1, arg2) and vec_any_le(arg1, arg2)

To implement the predicates for operand types vector float, supply the operands to the instruction in reverse order.

5.3.1.3 vec_all_lt(arg1, arg2) and vec_any_lt(arg1, arg2)

To implement the predicates for all operand types, supply the operands to the instruction in reverse order.

5.3.1.4 vec_all_nan(arg1) and vec_any_nan(arg1)

To implement the predicates, supply the operand to the instruction twice.

5.3.1.5 vec_all_nle(arg1, arg2) and vec_any_nle(arg1, arg2)

To implement the predicates, supply the operands to the instruction in reverse order.

5.3.1.6 vec_all_nlt(arg1, arg2) and vec_any_nlt(arg1, arg2)

To implement the predicates, supply the operands to the instruction in reverse order.

5.3.1.7 vec_all_numeric(arg1) and vec_any_numeric(arg1)

To implement the predicates, supply the operand to the instruction twice.

5.4 Debugger Implementation Notes

5.5 Coding Examples